

# Java Persistence

[Wikibooks.org](https://www.wikibooks.org/)

March 18, 2013

On the 28th of April 2012 the contents of the English as well as German Wikibooks and Wikipedia projects were licensed under Creative Commons Attribution-ShareAlike 3.0 Unported license. An URI to this license is given in the list of figures on page 283. If this document is a derived work from the contents of one of these projects and the content was still licensed by the project under this license at the time of derivation this document has to be licensed under the same, a similar or a compatible license, as stated in section 4b of the license. The list of contributors is included in chapter Contributors on page 281. The licenses GPL, LGPL and GFDL are included in chapter Licenses on page 287, since this book and/or parts of it may or may not be licensed under one or more of these licenses, and thus require inclusion of these licenses. The licenses of the figures are given in the list of figures on page 283. This PDF was generated by the L<sup>A</sup>T<sub>E</sub>X typesetting software. The L<sup>A</sup>T<sub>E</sub>X source code is included as an attachment (`source.7z.txt`) in this PDF file. To extract the source from the PDF file, we recommend the use of <http://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/> utility or clicking the paper clip attachment symbol on the lower left of your PDF Viewer, selecting **Save Attachment**. After extracting it from the PDF file you have to rename it to `source.7z`. To uncompress the resulting archive we recommend the use of <http://www.7-zip.org/>. The L<sup>A</sup>T<sub>E</sub>X source itself was generated by a program written by Dirk Hünninger, which is freely available under an open source license from [http://de.wikibooks.org/wiki/Benutzer:Dirk\\_Huenniger/wb2pdf](http://de.wikibooks.org/wiki/Benutzer:Dirk_Huenniger/wb2pdf). This distribution also contains a configured version of the `pdflatex` compiler with all necessary packages and fonts needed to compile the L<sup>A</sup>T<sub>E</sub>X source included in this PDF file.

# Contents

<b>1</b>	<b>Preface</b>	<b>3</b>
<b>2</b>	<b>What is Java persistence?</b>	<b>5</b>
<b>3</b>	<b>What is Java?</b>	<b>7</b>
3.1	Google Trends . . . . .	7
3.2	See also . . . . .	7
<b>4</b>	<b>What is a database?</b>	<b>9</b>
<b>5</b>	<b>What is JPA?</b>	<b>11</b>
<b>6</b>	<b>What is new in JPA 2.0?</b>	<b>13</b>
<b>7</b>	<b>Other Persistence Specs</b>	<b>15</b>
<b>8</b>	<b>Why use JPA or ORM?</b>	<b>17</b>
<b>9</b>	<b>Persistence Products</b>	<b>19</b>
9.1	Existing Persistence Products . . . . .	19
<b>10</b>	<b>EclipseLink</b>	<b>23</b>
<b>11</b>	<b>TopLink</b>	<b>25</b>
<b>12</b>	<b>Hibernate</b>	<b>27</b>
<b>13</b>	<b>TopLink Essentials</b>	<b>29</b>
<b>14</b>	<b>Kodo</b>	<b>31</b>
<b>15</b>	<b>Open JPA</b>	<b>33</b>
<b>16</b>	<b>Ebean</b>	<b>35</b>
16.1	Index . . . . .	35
<b>17</b>	<b>Mapping</b>	<b>37</b>
<b>18</b>	<b>Mapping</b>	<b>39</b>
18.1	Access Type . . . . .	42
18.2	Common Problems . . . . .	43
<b>19</b>	<b>Tables</b>	<b>45</b>

<b>20</b>	<b>Advanced</b>	<b>47</b>
20.1	Multiple tables . . . . .	47
20.2	Multiple tables with foreign keys . . . . .	49
20.3	Multiple table joins . . . . .	50
20.4	Multiple table outer joins . . . . .	51
20.5	Tables with special characters and mixed case . . . . .	52
20.6	Table qualifiers, schemas, or creators . . . . .	52
<b>21</b>	<b>Identity</b>	<b>55</b>
<b>22</b>	<b>Sequencing</b>	<b>57</b>
22.1	Sequence Strategies . . . . .	57
<b>23</b>	<b>Advanced</b>	<b>65</b>
23.1	Composite Primary Keys . . . . .	65
23.2	Primary Keys through OneToOne and ManyToOne Relationships . . . . .	69
23.3	Advanced Sequencing . . . . .	73
23.4	Primary Keys through Triggers . . . . .	74
23.5	Primary Keys through Events . . . . .	75
23.6	No Primary Key . . . . .	75
23.7	Single Table Inheritance . . . . .	77
23.8	Joined, Multiple Table Inheritance . . . . .	79
<b>24</b>	<b>Advanced</b>	<b>83</b>
24.1	Table Per Class Inheritance . . . . .	83
24.2	Mapped Superclasses . . . . .	84
<b>25</b>	<b>Embeddables</b>	<b>89</b>
<b>26</b>	<b>Advanced</b>	<b>93</b>
26.1	Sharing . . . . .	93
26.2	Embedded Ids . . . . .	94
26.3	Nulls . . . . .	94
26.4	Nesting . . . . .	95
26.5	Inheritance . . . . .	96
26.6	Relationships . . . . .	96
26.7	Collections . . . . .	97
26.8	Querying . . . . .	98
<b>27</b>	<b>Locking</b>	<b>99</b>
27.1	Optimistic Locking . . . . .	99
27.2	Advanced . . . . .	104
<b>28</b>	<b>Basics</b>	<b>111</b>
28.1	Common Problems . . . . .	113
<b>29</b>	<b>Advanced</b>	<b>115</b>
29.1	Temporal, Dates, Times, Timestamps and Calendars . . . . .	115
29.2	Enums . . . . .	117

29.3	LOBs, BLOBs, CLOBs and Serialization . . . . .	118
29.4	Lazy Fetching . . . . .	119
29.5	Optional . . . . .	119
29.6	Column Definition and Schema Generation . . . . .	120
29.7	Insertable, Updatable / Read Only Fields / Returning . . . . .	121
29.8	Conversion . . . . .	122
29.9	Custom Types . . . . .	122
<b>30</b>	<b>Relationships</b>	<b>125</b>
30.1	Lazy Fetching . . . . .	126
30.2	Cascading . . . . .	129
30.3	Orphan Removal (JPA 2.0) . . . . .	130
30.4	Target Entity . . . . .	131
30.5	Collections . . . . .	132
30.6	Common Problems . . . . .	135
<b>31</b>	<b>Advanced</b>	<b>139</b>
31.1	Advanced Relationships . . . . .	139
31.2	Maps . . . . .	140
31.3	Join Fetching . . . . .	145
31.4	Batch Fetching . . . . .	147
31.5	Filtering, Complex Joins . . . . .	148
31.6	Variable and Heterogeneous Relationships . . . . .	148
31.7	Nested Collections, Maps and Matrices . . . . .	149
<b>32</b>	<b>OneToOne</b>	<b>151</b>
32.1	Inverse Relationships, Target Foreign Keys and Mapped By . . . . .	154
32.2	See Also . . . . .	155
32.3	Common Problems . . . . .	155
<b>33</b>	<b>Advanced</b>	<b>157</b>
33.1	Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys	157
33.2	Mapping a OneToOne Using a Join Table . . . . .	161
<b>34</b>	<b>ManyToOne</b>	<b>165</b>
34.1	See Also . . . . .	167
34.2	Common Problems . . . . .	167
<b>35</b>	<b>Advanced</b>	<b>169</b>
35.1	Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys	169
<b>36</b>	<b>OneToMany</b>	<b>171</b>
36.1	Join Table . . . . .	174
36.2	See Also . . . . .	176
36.3	Common Problems . . . . .	176
<b>37</b>	<b>Advanced</b>	<b>177</b>
37.1	Unidirectional OneToMany, No Inverse ManyToOne, No Join Table ( <b>JPA 2.0 ONLY</b> ) . . . . .	177

<b>38</b>	<b>ManyToMany</b>	<b>179</b>
38.1	Bi-directional Many to Many . . . . .	180
38.2	See Also . . . . .	181
38.3	Common Problems . . . . .	182
<b>39</b>	<b>Advanced</b>	<b>183</b>
39.1	Mapping a Join Table with Additional Columns . . . . .	183
<b>40</b>	<b>ElementCollection</b>	<b>187</b>
40.1	Embedded Collections . . . . .	188
40.2	Basic Collections . . . . .	189
40.3	See Also . . . . .	191
40.4	Common Problems . . . . .	191
<b>41</b>	<b>Advanced Topics</b>	<b>193</b>
41.1	Events . . . . .	193
41.2	Views . . . . .	196
41.3	Interfaces . . . . .	197
41.4	Stored Procedures . . . . .	198
41.5	Structured Object-Relational Data Types . . . . .	199
41.6	XML Data Types . . . . .	200
41.7	Filters . . . . .	201
41.8	History . . . . .	202
41.9	Logical Deletes . . . . .	203
41.10	Auditing . . . . .	203
41.11	Replication . . . . .	203
41.12	Partitioning . . . . .	204
41.13	Data Integration . . . . .	204
41.14	NoSQL (and EIS, legacy, XML, and non-relational data) . . . . .	204
41.15	Multi-Tenancy . . . . .	205
41.16	Dynamic Data . . . . .	205
<b>42</b>	<b>Runtime</b>	<b>207</b>
<b>43</b>	<b>Entity Manager</b>	<b>209</b>
43.1	Java Standard Edition . . . . .	209
43.2	Java Enterprise Edition . . . . .	211
<b>44</b>	<b>Querying</b>	<b>215</b>
44.1	Named Queries . . . . .	216
44.2	Dynamic Queries . . . . .	217
44.3	JPQL . . . . .	218
44.4	Parameters . . . . .	218
44.5	Query Results . . . . .	219
44.6	Common Queries . . . . .	220
<b>45</b>	<b>Advanced</b>	<b>225</b>
45.1	Join Fetch and Query Optimization . . . . .	225
45.2	Timeouts, Fetch Size and other JDBC Optimizations . . . . .	225

---

45.3	Update and Delete Queries . . . . .	226
45.4	Flush Mode . . . . .	226
45.5	Pagination, Max/First Results . . . . .	227
45.6	Native SQL Queries . . . . .	229
45.7	Stored Procedures . . . . .	231
45.8	Raw JDBC . . . . .	231
<b>46</b>	<b>JPQL BNF</b>	<b>233</b>
46.1	Select . . . . .	233
46.2	Update . . . . .	237
46.3	Delete . . . . .	238
46.4	Literals . . . . .	238
46.5	New in JPA 2.0 . . . . .	239
<b>47</b>	<b>Persisting</b>	<b>241</b>
47.1	Persist . . . . .	241
47.2	Merge . . . . .	243
47.3	Remove . . . . .	245
<b>48</b>	<b>Advanced</b>	<b>247</b>
48.1	Refresh . . . . .	247
48.2	Lock . . . . .	247
48.3	Get Reference . . . . .	248
48.4	Flush . . . . .	248
48.5	Clear . . . . .	249
48.6	Close . . . . .	250
48.7	Get Delegate . . . . .	250
48.8	Unwrap (JPA 2.0) . . . . .	251
<b>49</b>	<b>Transactions</b>	<b>253</b>
49.1	Resource Local Transactions . . . . .	253
49.2	JTA Transactions . . . . .	254
<b>50</b>	<b>Advanced</b>	<b>257</b>
50.1	Join Transaction . . . . .	257
50.2	Retrying Transactions, Handling Commit Failures . . . . .	258
50.3	Nested Transactions . . . . .	258
50.4	Transaction Isolation . . . . .	259
<b>51</b>	<b>Caching</b>	<b>261</b>
51.1	Object Identity . . . . .	262
51.2	Object Cache . . . . .	262
51.3	Data Cache . . . . .	263
51.4	Cache Types . . . . .	264
51.5	Query Cache . . . . .	265
51.6	Stale Data . . . . .	265
51.7	Cache Transaction Isolation . . . . .	271
51.8	Common Problems . . . . .	271

<b>52 Spring</b>	<b>273</b>
52.1 Persistence . . . . .	273
52.2 JPA . . . . .	273
<b>53 Databases</b>	<b>275</b>
<b>54 MySQL</b>	<b>277</b>
54.1 Installing . . . . .	277
54.2 Configuration tips . . . . .	277
<b>55 References</b>	<b>279</b>
<b>56 Contributors</b>	<b>281</b>
<b>List of Figures</b>	<b>283</b>
<b>57 Licenses</b>	<b>287</b>
57.1 GNU GENERAL PUBLIC LICENSE . . . . .	287
57.2 GNU Free Documentation License . . . . .	288
57.3 GNU Lesser General Public License . . . . .	289





# 1 Preface



## 1.0.1 What is this book about?

This book is meant to cover Java persistence, that is, storing stuff in the Java programming language to a persistent storage medium. Specifically using the Java Persistence API (JPA) to store Java objects to relational databases, but I would like it to have a somewhat wider scope than just JPA and concentrate more on general persistence patterns and use cases, after all JPA is just the newest of many failed Java persistence standards, this book should be able to evolve beyond JPA when it is replaced by the next persistence standard. I do not want this to be just a regurgitation of the JPA Spec, nor a User Manual to using one of the JPA products, but more focused on real-world use cases of users and applications trying to make use of JPA (or other Java persistence solution) and the patterns they evolved and pitfalls they made.



## 1.0.2 Intended Audience

This book is intended to be useful for or to anyone learning to, or developing Java applications that require persisting data to a database. It is mainly intended for Java developers intending to persist Java objects through the Java Persistence API (JPA) standard to a relational database. Please don't just read this book, if you're learning or developing with JPA please contribute your experiences to this book.



## 1.0.3 Style

This book is meant to be written in a casual manner. The goal is avoid sounding dry, overly technical, or impersonal. The book should sound casual, like a co-worker explaining to you how to use something, or a fellow consultant relating their latest engagement to another. Please refrain from being overly critical of any product, ranting about bugs, or marketing your own product or services.



### 1.0.4 Authors

Everyone is encouraged to participate in the ongoing development of this book. You do not need to be a Java persistence superstar to contribute, many times the best advice/information for other users comes from first time users who have not yet been conditioned to think something that may be confusing is obvious.

List of authors: (please contribute and sign your name)

James Sutherland<sup>1</sup> : Currently working on Oracle<sup>2</sup> TopLink<sup>3</sup> and Eclipse<sup>4</sup> EclipseLink<sup>5</sup>, over 12 years of experience in object persistence and ORM.

Doug Clarke<sup>6</sup> : Oracle<sup>7</sup> TopLink<sup>8</sup> and Eclipse<sup>9</sup> EclipseLink<sup>10</sup>, over 10 years of experience in the object persistence industry.

Preface<sup>11</sup>

---

1 <http://en.wikibooks.org/wiki/User%3AJamesssss>  
2 <http://en.wikipedia.org/wiki/Oracle%20Corporation>  
3 <http://en.wikipedia.org/wiki/TopLink>  
4 <http://en.wikipedia.org/wiki/Eclipse%20Foundation>  
5 <http://en.wikipedia.org/wiki/EclipseLink>  
6 <http://en.wikibooks.org/wiki/User%3ADjclarke>  
7 <http://en.wikipedia.org/wiki/Oracle%20Corporation>  
8 <http://en.wikipedia.org/wiki/TopLink>  
9 <http://en.wikipedia.org/wiki/Eclipse%20Foundation>  
10 <http://en.wikipedia.org/wiki/EclipseLink>  
11 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

## 2 What is Java persistence?

**Persistence**<sup>1</sup>, in computer science, is an adjective describing data that outlives the process that created it. Java persistence could be defined as storing anything to any level of persistence using the Java programming language, but obviously this would be too broad a definition to cover in a single book. That is why this book is more focused on storing Java objects<sup>2</sup> to relational databases<sup>3</sup>. In particular using the Java Persistence API<sup>4</sup> (JPA).

There are many ways to make data persist in Java, including (to name a few): JDBC<sup>5</sup>, serialization<sup>6</sup>, file IO, JCA<sup>7</sup>, object databases<sup>8</sup>, and XML databases<sup>9</sup>. However, the majority of data is persisted in databases, specifically relational databases. Most things that you do on a computer or web site that involve storing data, involve accessing a relational database. Relational databases are the standard mode of persistent storage for most industries, from banking to manufacturing.

There are many things that can be stored in databases with Java. Java data includes strings, numbers, date and byte arrays, images, XML, and Java objects. Many Java applications use Java objects to model their application data. Because Java is an Object Oriented<sup>10</sup> language, storing Java objects is a natural and common approach to persisting data from Java.

There are many ways to access a relational database from Java, JPA is just the latest of many different specifications, but it seems to be the direction that most programmers are heading.

What is Java persistence<sup>11</sup> What is Java persistence<sup>12</sup>

- 
- 1 [http://en.wikipedia.org/wiki/Persistence\\_%28computer\\_science%29](http://en.wikipedia.org/wiki/Persistence_%28computer_science%29)
  - 2 [http://en.wikipedia.org/wiki/Object%20%28computer\\_science%29%20](http://en.wikipedia.org/wiki/Object%20%28computer_science%29%20)
  - 3 [http://en.wikipedia.org/wiki/Relational\\_databases%20](http://en.wikipedia.org/wiki/Relational_databases%20)
  - 4 [http://en.wikipedia.org/wiki/Java\\_Persistence\\_API%20](http://en.wikipedia.org/wiki/Java_Persistence_API%20)
  - 5 <http://en.wikipedia.org/wiki/JDBC%20>
  - 6 <http://en.wikipedia.org/wiki/Serialization%20>
  - 7 [http://en.wikipedia.org/wiki/J2EE\\_Connector\\_Architecture%20](http://en.wikipedia.org/wiki/J2EE_Connector_Architecture%20)
  - 8 [http://en.wikipedia.org/wiki/Object\\_databases%20](http://en.wikipedia.org/wiki/Object_databases%20)
  - 9 [http://en.wikipedia.org/wiki/XML\\_database%20](http://en.wikipedia.org/wiki/XML_database%20)
  - 10 <http://en.wikipedia.org/wiki/Object%20oriented%20>
  - 11 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FWhat%20is%20Java%20persistence>
  - 12 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>



## 3 What is Java?

Java<sup>1</sup> is an object oriented<sup>2</sup> programming language first released by Sun Microsystems<sup>3</sup> in 1995. It blended concepts from existing languages such as C++<sup>4</sup> and Smalltalk<sup>5</sup> into a new programming language. It achieved its success over the many rival languages of the day because it was associated with this newish thing called the "Internet" in allowing Java applets to be embedded in web pages and run using Netscape. Its other main reason for success was unlike many of its competitors it was open, "free", and not embedded with a integrated development environment (IDE). Java also included the source code to its class library. This enabled Java to be adopted by many different companies producing their own Java development environments but sharing the same language, this open model fostered the growth in the Java language and continues today with the open sourcing of Java.

Java quickly moved from allowing developers to build dinky applets to being the standard server-side language running much of the Internet today. The Enterprise Edition (JEE) of Java was defined to provide an open model for server application to be written and portable across any compliant JEE platform provider. The JEE standard is basically a basket of other Java specifications brought together under one umbrella and has major providers including IBM WebSphere, RedHat JBoss, Sun Glassfish, BEA WebLogic, Oracle AS and many others.

### 3.1 Google Trends

- Programming Languages<sup>67</sup>
- JEE Servers<sup>89</sup>

### 3.2 See also

- Java Programming<sup>10</sup>

---

1 <http://en.wikipedia.org/wiki/Java%20%28programming%20language%29%20>  
2 <http://en.wikipedia.org/wiki/object%20oriented%20>  
3 <http://en.wikipedia.org/wiki/Sun%20Microsystems%20>  
4 <http://en.wikipedia.org/wiki/C%2B%2B%20>  
5 <http://en.wikipedia.org/wiki/Smalltalk%20>  
6 <http://www.google.com/trends?q=c%23%2C+php%2C+java%2C+c%2B%2B%2C+perl&ctab=0&geo=all&date=all&sort=0>  
7 Unfortunately hard to seperate Java island from Java language.  
8 <http://www.google.com/trends?q=websphere%2C+weblogic%2C+jboss%2C+glassfish%2C+geronimo+apache&ctab=0&geo=all&date=all&sort=0>  
9 Could not include Oracle because of Oracle database hits.  
10 <http://en.wikibooks.org/wiki/Java%20Programming>

What is Java<sup>11</sup> What is Java<sup>12</sup>

- 
- 11 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FWhat%20is%20Java%20persistence>
- 12 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

## 4 What is a database?

A database<sup>1</sup> is a program that stores data. There are many types of databases: flat-file, hierarchical, relational, object-relational, object-oriented, xml, and others. The original databases were mainly proprietary and non-standardized.

Relational databases were the first databases to achieve great success and standardization. Relational databases are characterized by the SQL<sup>2</sup> (structured query language) standard to query and modify the database, their client/server<sup>3</sup> architecture, and relational table storage structure. Relational databases achieved great success because their standardization allowed many different vendors such as Oracle<sup>4</sup>, IBM<sup>5</sup>, and Sybase<sup>6</sup> to produce interoperable products giving users the flexibility to switch their vendor and avoid vendor lock-in to a proprietary solution. Their client/server architecture allows the client programming language to be decoupled from the server, allowing the database server to support interface APIs into multiple different programming languages and clients.

Although relational databases are relatively old technology, they still dominate the industry. There have been many attempts to replace the relational model, first with object-oriented databases, then with object-relational databases, and finally with XML databases, but none of the new database models achieved much success and relational databases remain the overwhelmingly dominant database model.

The main relational databases used today are: Oracle<sup>7</sup>, MySQL<sup>8</sup> (Oracle), PostgreSQL<sup>9</sup>, DB2<sup>10</sup> (IBM), SQL Server<sup>11</sup> (Microsoft).

- Google trend for databases<sup>12</sup>

What is a database<sup>13</sup> What is a database<sup>14</sup>

---

1 <http://en.wikipedia.org/wiki/database>  
2 <http://en.wikipedia.org/wiki/SQL>  
3 <http://en.wikipedia.org/wiki/client%2Fserver>  
4 <http://en.wikipedia.org/wiki/Oracle%20Corporation>  
5 <http://en.wikipedia.org/wiki/IBM>  
6 <http://en.wikipedia.org/wiki/Sybase>  
7 <http://en.wikipedia.org/wiki/Oracle%20database>  
8 <http://en.wikipedia.org/wiki/MySQL>  
9 <http://en.wikipedia.org/wiki/PostgreSQL>  
10 <http://en.wikipedia.org/wiki/DB2>  
11 <http://en.wikipedia.org/wiki/Microsoft%20SQL%20Server>  
12 <http://www.google.com/trends?q=oracle%2C+sybase%2C+sql+server%2C+mysql%2C+db2&ctab=0&geo=all&date=all&sort=0>  
13 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FWhat%20is%20Java%20Persistence>  
14 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>





## 5 What is JPA?

The Java Persistence Architecture API (JPA<sup>1</sup>) is a Java specification for accessing, persisting, and managing data between Java objects / classes and a relational database. JPA was defined as part of the EJB<sup>2</sup> 3.0 specification as a replacement for the EJB 2 CMP Entity Beans specification. JPA is now considered the standard industry approach for Object to Relational Mapping (ORM) in the Java Industry.

JPA itself is just a specification, not a product; it cannot perform persistence or anything else by itself. JPA is just a set of interfaces, and requires an implementation. There are open-source and commercial JPA implementations to choose from and any Java EE 5 application server should provide support for its use. JPA also requires a database to persist to.

JPA allows POJO<sup>3</sup> (Plain Old Java Objects) to be easily persisted without requiring the classes to implement any interfaces or methods as the EJB 2 CMP specification required. JPA allows an object's object-relational mappings to be defined through standard annotations or XML defining how the Java class maps to a relational database table. JPA also defines a runtime EntityManager API for processing queries and transaction on the objects against the database. JPA defines an object-level query language, JPQL, to allow querying of the objects from the database.

JPA is the latest of several Java persistence specifications. The first was the OMG persistence service Java binding, which was never very successful; I'm not sure of any commercial products supporting it. Next came EJB 1.0 CMP Entity Beans, which was very successful in being adopted by the big Java EE providers (BEA, IBM), but there was a backlash against the spec by some users who thought the spec requirements on the Entity Beans overly complex and overhead and performance poor. EJB 2.0 CMP tried to reduce some of the complexity of Entity Beans through introducing local interfaces, but the majority of the complexity remained. EJB 2.0 also lacked portability, in that the deployment descriptors defining the object-relational mapping were not specified and were all proprietary.

This backlash in part led to the creation of another Java persistence specification, JDO<sup>4</sup> (Java Data Objects). JDO obtained somewhat of a "cult" following of several independent vendors such as Kodo JDO, and several open-source implementations, but never had much success with the big Java EE vendors.

Despite the two competing Java persistence standards of EJB CMP and JDO, the majority of users continued to prefer proprietary api solutions, mainly TopLink<sup>5</sup> (which had been

---

1 <http://en.wikipedia.org/wiki/Java%20Persistence%20API>

2 <http://en.wikipedia.org/wiki/EJB>

3 <http://en.wikipedia.org/wiki/POJO>

4 <http://en.wikipedia.org/wiki/Java%20Data%20Objects>

5 Chapter 11 on page 25

around for some time and had its own POJO API) and Hibernate<sup>6</sup> (which was a relatively new open-source product that also had its own POJO API and was quickly becoming the open-source industry standard). The TopLink product formerly owned by WebGain was also acquired by Oracle, increasing its influence on the Java EE community.

The EJB CMP backlash was only part of a backlash against all of Java EE which was seen as too complex in general and prompted such products as the Spring container. This led the EJB 3.0 specification to have a main goal of reducing the complexity, which led the spec committee down the path of JPA. JPA was meant to unify the EJB 2 CMP, JDO, Hibernate, and TopLink APIs and products, and seems to have been very successful in doing so.

Currently most of the persistence vendors have released implementations of JPA confirming its adoption by the industry and users. These include Hibernate (acquired by JBoss, acquired by Red Hat), TopLink (acquired by Oracle), and Kodo JDO (acquired by BEA, acquired by Oracle). Other products that have added support for JPA include Cocobase (owned by Thought Inc.) and JPOX.

- EJB JPA Spec<sup>7</sup>
- JPA ORM XML Schema<sup>8</sup>
- JPA Persistence XML Schema<sup>9</sup>
- JPA JavaDoc<sup>10</sup>
- JPQL BNF<sup>11</sup>

What is JPA<sup>12</sup> What is JPA<sup>13</sup>

---

6 Chapter 12 on page 27

7 <http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html>

8 [http://java.sun.com/xml/ns/persistence/orm\\_1\\_0.xsd](http://java.sun.com/xml/ns/persistence/orm_1_0.xsd)

9 [http://java.sun.com/xml/ns/persistence/persistence\\_1\\_0.xsd](http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd)

10 <https://java.sun.com/javase/5/docs/api/javax/persistence/package-summary.html>

11 Chapter 45.8 on page 232

12 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FWhat%20is%20Java%20Persistence>

13 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

## 6 What is new in JPA 2.0?

The JPA 2.0 specification adds several enhancements to the JPA 1.0 specification including:

- Extended Map support<sup>1</sup> - Support for maintaining a key column for a **Basic**, **Embeddable**, or **Entity** key value in any collection relationship using a **Map**.
- Derived Identifiers<sup>2</sup>
- Nested embedding<sup>3</sup>
- New collection mappings<sup>4</sup> - Support for collections of **Basic** or **Embeddable** types.
- Undirectional OneToMany<sup>5</sup>
- Ordered List mappings<sup>6</sup> - Support for maintaining an index column in any collection relationship using a **List**.
- Orphan removal<sup>7</sup> - Automatic deletion of objects removed from relationships.
- Pessimistic Locking<sup>8</sup>
- EntityManager API updates<sup>9</sup>
- Cache APIs<sup>10</sup>
- Standard Properties<sup>11</sup>
- Metadata
- Criteria API<sup>12</sup>
- JPQL enhancements<sup>13</sup>

### 6.0.1 Resources

- JPA 2.0 Spec<sup>14</sup>
- JPA 2.0 Reference Implementation (EclipseLink)<sup>15</sup>
- Eclipse EclipseLink to be JPA 2.0 Reference Implementation<sup>16</sup>
- JPA 2.0 Examples<sup>17</sup>

---

1 Chapter 31.2.1 on page 141

2 Chapter 23.2.2 on page 71

3 Chapter 26.4 on page 95

4 Chapter 39.1.2 on page 185

5 Chapter 37.1 on page 177

6 Chapter 30.5.4 on page 133

7 Chapter 31.6 on page 148

8 Chapter 27.2.7 on page 107

9 Chapter 46.5 on page 240

10 Chapter 51.6.4 on page 268

11 Chapter 43.1 on page 209

12 Chapter 44.2.1 on page 218

13 Chapter 46.5 on page 239

14 <http://jcp.org/en/jsr/detail?id=317>

15 <http://www.eclipse.org/eclipselink>

16 [http://www.eclipse.org/org/press-release/20080317\\_Eclipselink.php](http://www.eclipse.org/org/press-release/20080317_Eclipselink.php)

17 [http://wiki.eclipse.org/EclipseLink/Examples/JPA#JPA\\_2.0](http://wiki.eclipse.org/EclipseLink/Examples/JPA#JPA_2.0)

What is new in JPA 2.0<sup>18</sup> What is new in JPA 2.0<sup>19</sup>

---

<sup>18</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FWhat%20is%20Java%20persistence>  
<sup>19</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

## 7 Other Persistence Specs

There are many specifications related to persistence in Java. The following table summarises the latest version of each specification.<sup>1</sup>

Spec	Version	Year of Last Release   -
JDBC <sup>2</sup> (Java DataBase Connectivity)		
	5.0	2010
JDO <sup>3</sup> (Java Data Objects)		
	3.0	2010
JPA <sup>4</sup> (Java Persistence API)		
	2.0	2009
JCA <sup>5</sup> (Java EE Connector Architecture)		
	1.6	2009
SDO <sup>6</sup> (Service Data Objects)		
	2.1	2006
EJB CMP <sup>7</sup> (Enterprise Java Beans, Container Managed Persistence)		
	2.1 (EJB)	2003

Other Persistence Specs<sup>8</sup> Other Persistence Specs<sup>9</sup>

---

<sup>1</sup> Last updated 2010-10  
<sup>2</sup> <http://en.wikipedia.org/wiki/JDBC>  
<sup>3</sup> <http://en.wikipedia.org/wiki/Java%20Data%20Objects>  
<sup>4</sup> <http://en.wikipedia.org/wiki/Java%20Persistence%20API>  
<sup>5</sup> <http://en.wikipedia.org/wiki/Java%20EE%20Connector%20Architecture>  
<sup>6</sup> <http://en.wikipedia.org/wiki/Service%20Data%20Objects>  
<sup>7</sup> <http://en.wikipedia.org/wiki/EJB>  
<sup>8</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FWhat%20is%20Java%20persistence>  
<sup>9</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>



## 8 Why use JPA or ORM?

This is an intriguing question. There are many reasons to use an ORM framework or persistence product, and many reasons to use JPA in particular.

### Reasons for ORM

- Leverages large persistence library to avoid developing solutions to problems that others have already solved.
- Avoids low level JDBC and SQL code.
- Leverages object oriented programming and object model usage.
- Provides database and schema independence.
- Most ORM products are free and open source.
- Many enterprise corporations provide support and services for ORM products.
- Provides high end performance features such as caching and sophisticated database and query optimizations.

### Reasons for JPA

- It is a standard and part of EJB3 and JEE.
- Many free and open source products with enterprise level support.
- Portability across application servers and persistence products (avoids vendor lock-in).
- A usable and functional specification.
- Supports both JEE and JSE.

ORM can be a hot topic for some people, and there are many ORM camps. There are those that endorse a particular standard or product. There are those that don't believe in ORM or even objects in general and prefer JDBC. There are those that still believe that object databases are the way to go. Personally I would recommend you use whatever technology you are most comfortable with, but if you have never used ORM or JPA, perhaps give it a try and see if you like it. The below list provides several discussions on why or why not to use JPA and ORM.

### Discussions on JPA and ORM Usage

- Why do we need anything other than JDBC? (Java Ranch)<sup>1</sup>
- JPA Explained (The Server Side)<sup>2</sup>

---

1 [http://saloon.javaranch.com/cgi-bin/ubb/ultimatebb.cgi?ubb=get\\_topic&f=78&t=003738](http://saloon.javaranch.com/cgi-bin/ubb/ultimatebb.cgi?ubb=get_topic&f=78&t=003738)

2 [http://www.theserverside.com/news/thread.tss?thread\\_id=44526](http://www.theserverside.com/news/thread.tss?thread_id=44526)



- JPA vs JDO (The Server Side)<sup>3</sup>

Why use JPA or ORM<sup>4</sup> Why use JPA or ORM<sup>5</sup>

---

<sup>3</sup> [http://www.theserverside.com/news/thread.tss?thread\\_id=40965](http://www.theserverside.com/news/thread.tss?thread_id=40965)

<sup>4</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FWhat%20is%20Java%20persistence>

<sup>5</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

## 9 Persistence Products

There are many persistence products to choose from. Most persistence products now support a JPA interface, although there still are some exceptions. Which product you use depends on your preference, but most people would recommend you use the JPA standard whichever product you choose. This gives you the flexibility to switch persistence providers, or port your application to another server platform which may use a different persistence provider.

Determining which persistence product to use involves many criteria. Valid things to consider include:

- Which persistence product does your server platform support and integrate with?
- What is the cost of the product, is it free and open source, can you purchase enterprise level support and services?
- Do you have an existing relationship with the company producing the product?
- Is the product active and does it have a large user base?
- How does the product perform and scale?
- Does the product integrate with your database platform?
- Does the product have active and open forums, do questions receive useful responses?
- Is the product JPA compliant, what functionality does the product offer beyond the JPA specification?

### 9.1 Existing Persistence Products

The following table summaries existing persistence products.<sup>1</sup>

---

<sup>1</sup> Last updated 2013-01

Product	JPA 1.0	JPA 2.0	JDO 2.0	JDO 3.0	CMP 2.1	Version	Year of Last Release	Open Source	Application Servers <sup>2</sup>
Hibernate <sup>3</sup> (Red Hat)	Yes	Yes				4.1	2012	Yes	JBoss
EclipseLink <sup>4</sup> (Eclipse)	Yes	Yes						Yes	Oracle Weblogic (12c), Glassfish (v3)
TopLink <sup>5</sup> (Oracle)	Yes	Yes			Yes				Oracle Weblogic (12c), OracleAS (10.1.3)

---

3 Chapter 12 on page 27  
4 Chapter 10 on page 23  
5 Chapter 11 on page 25

Product	JPA 1.0	JPA 2.0	JDO 2.0	JDO 3.0	CMP 2.1	Version	Year of Last Release	Open Source	Application Servers <sup>2</sup>
Open-JPA <sup>6</sup> (Apache)	Yes	Yes				2.1.0	2011	Yes	Gerontimo, WebSphere Application Server (8.0)
DataNucleus <sup>7</sup> (DataNucleus)	Yes	Yes	Yes	Yes		3.1.3	2013	Yes	
TopLink Essentials <sup>8</sup> (java.net)	Yes					2.0	2007	Yes	Glassfish (v2), SunAS (9), OracleAS (10.1.3)
Kodo <sup>9</sup> (Oracle)	Yes		Yes			4.1	2007		Oracle WebLogic (10.3)

<sup>6</sup> Chapter 15 on page 33  
<sup>7</sup> <http://www.datanucleus.org/>  
<sup>8</sup> Chapter 13 on page 29  
<sup>9</sup> Chapter 14 on page 31

Persistence Products<sup>10</sup> Persistence Products<sup>11</sup>

---

<sup>10</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FPersistence%20Products>  
<sup>11</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

## 10 EclipseLink

EclipseLink<sup>1</sup> is the open source Eclipse Persistence Services Project from the Eclipse Foundation. The product provides an extensible framework that allows Java developers to interact with various data services, including databases, XML, and Enterprise Information Systems (EIS). EclipseLink supports a number of persistence standards including the Java Persistence API (JPA), Java API for XML Binding (JAXB), Java Connector Architecture (JCA), and Service Data Objects (SDO).

EclipseLink is based on the TopLink<sup>2</sup> product, which Oracle<sup>3</sup> contributed the source code from to create the EclipseLink project. The original contribution was from TopLink's 11g code base, and the entire code-base/feature set was contributed, with only EJB 2 CMP and some minor Oracle AS specific integration removed. This differs from the TopLink Essentials Glassfish contribution, which did not include some key enterprise features. The package names were changed and some of the code was moved around.

The TopLink Mapping Workbench UI has also been contributed to the project.

EclipseLink is the intended path forward for persistence for Oracle and TopLink. It is intended that the next major release of Oracle TopLink will include EclipseLink as well as the next major release of Oracle AS.

EclipseLink supports usage in an OSGi<sup>4</sup> environment.

EclipseLink was announced to be the JPA 2.0 reference implementation, and announced to be the JPA provider for Glassfish v3.

- EclipseLink Home<sup>5</sup>
- EclipseLink Newsgroup<sup>6</sup>
- EclipseLink Wiki<sup>7</sup>

EclipseLink<sup>8</sup> EclipseLink<sup>9</sup>

---

1 <http://en.wikipedia.org/wiki/EclipseLink>  
2 Chapter 11 on page 25  
3 <http://en.wikipedia.org/wiki/Oracle%20Corporation>  
4 <http://en.wikipedia.org/wiki/OSGi>  
5 <http://www.eclipse.org/eclipselink/>  
6 <http://www.eclipse.org/newsportal/thread.php?group=eclipse.technology.eclipselink>  
7 <http://wiki.eclipse.org/EclipseLink>  
8 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FPersistence%20Products>  
9 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>



# 11 TopLink

TopLink<sup>1</sup> is one of the leading Java persistence products and JPA implementations. TopLink is produced by Oracle<sup>2</sup> and part of Oracle's OracleAS, WebLogic, and OC4J servers.

As of TopLink 11g, TopLink bundles the open source project EclipseLink<sup>3</sup> for most of its functionality.

The TopLink 11g release supports the JPA 1.0 specification. TopLink 10.1.3 also supports EJB CMP and is the persistence provider for OracleAS OC4J 10.1.3 for both JPA and EJB CMP. TopLink provides advanced object-relational mapping functionality beyond the JPA specification, as well as providing persistence for object-relational data-types, and Enterprise Information Systems (EIS/mainframes). TopLink includes sophisticated object caching and performance features. TopLink provides a Grid extension that integrate with Oracle Coherence. TopLink provides object-XML mapping support and provides a JAXB implementation and web service integration. TopLink provides a Service Data Object (SDO) implementation.

TopLink provides a rich user interface through the TopLink Mapping Workbench. The Mapping Workbench allows for graphical mapping of an object model to a data model, as allows for generation of a data model from an object model, and generation of an object model from a data model, and auto-mapping of an existing object and data model. The TopLink Mapping Workbench functionality is also integrated with Oracle's JDeveloper IDE.

TopLink contributed part of its source code to become the JPA 1.0 reference implementation under the Sun java.net Glassfish project. This open-source product is called TopLink Essentials, and despite a different package name (oracle.toplink.essentials) it is basically a branch of the source code of the TopLink product with some advanced functionality stripped out.

TopLink contributed practically its entire source code to the Eclipse Foundation EclipseLink product. This is an open source product currently in incubation that represents the path forward for TopLink. The package name is different (org.eclipse.persistence) but the source code it basically a branch of the TopLink 11g release. Oracle also contributed its Mapping Workbench source code to the project. The TopLink Mapping Workbench developers also were major contributors to the Eclipse Dali project for JPA support.

TopLink was first developed in Smalltalk and ported to Java in the 90's, and has over 15 years worth of object persistence solutions. TopLink originally provided a proprietary POJO persistence API, when EJB was first released TopLink provided one of the most popular EJB CMP implementations, although it continued to recommend its POJO solution.

---

<sup>1</sup> <http://en.wikipedia.org/wiki/TopLink>

<sup>2</sup> <http://en.wikipedia.org/wiki/Oracle%20Corporation>

<sup>3</sup> Chapter 10 on page 23



TopLink also provided a JDO 1.0 implementation for a few releases, but this was eventually deprecated and removed once the JPA specification had been formed. Oracle and TopLink have been involved in each of the EJB, JDO and EJB3/JPA expert groups, and Oracle was the co-lead for the EJB3/JPA specification.

- Oracle TopLink Home<sup>4</sup>
- Oracle TopLink Forum<sup>5</sup>
- Oracle TopLink Wiki<sup>6</sup>

### TopLink Resources

- TopLink Automatic Schema Generation Options<sup>7</sup>

TopLink<sup>8</sup> TopLink<sup>9</sup>

---

<sup>4</sup> <http://www.oracle.com/technology/products/ias/toplink/index.html>

<sup>5</sup> <http://forums.oracle.com/forums/forum.jspa?forumID=48>

<sup>6</sup> <http://wiki.oracle.com/page/TopLink>

<sup>7</sup> <http://docs.sun.com/app/docs/doc/819-3672/gbwmk?a=view>

<sup>8</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FPersistence%20Products>

<sup>9</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

## 12 Hibernate

Hibernate<sup>1</sup> was an open source project developed by a team of Java software developers around the world led by Gavin King. JBoss, Inc. (now part of Red Hat) later hired the lead Hibernate developers and worked with them in supporting Hibernate.

The current version of Hibernate is Version 4.1.x. Hibernate provides both a proprietary POJO API, and JPA support.

Hibernate <sup>2</sup> Hibernate <sup>3</sup>

---

<sup>1</sup> <http://en.wikipedia.org/wiki/Hibernate%20%28Java%29>

<sup>2</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FPersistence%20Products>

<sup>3</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>



## 13 TopLink Essentials

TopLink Essentials<sup>1</sup> is an open source project from the Sun java.net Glassfish community. It is the EJB3 JPA 1.0 reference implementation, and is the JPA provider for the Sun Glassfish v1 application server.

TopLink Essentials was based on the TopLink<sup>2</sup> product, which Oracle<sup>3</sup> contributed some of the source code from to create the TopLink Essentials project. The original contribution was from TopLink's 10.1.3 code base, only some of the TopLink product source code was contributed, which did not include some key enterprise features. The package names were changed and some of the code was moved around.

TopLink Essentials has been replaced by the EclipseLink<sup>4</sup> project. EclipseLink will be the JPA 2.0 reference implementation and be part of Sun Glassfish v3.

- TopLink Essentials Home<sup>5</sup>
- TopLink Essentials Forum<sup>6</sup>
- TopLink Essentials Wiki<sup>7</sup>

TopLink Essentials<sup>8</sup> TopLink Essentials<sup>9</sup>

---

1 <http://en.wikipedia.org/wiki/TopLink>  
2 Chapter 11 on page 25  
3 <http://en.wikipedia.org/wiki/Oracle%20Corporation>  
4 Chapter 10 on page 23  
5 <https://glassfish.dev.java.net/javaee5/persistence/index.html>  
6 <http://www.nabble.com/java.net---glassfish-persistence-f13455.html>  
7 <http://wiki.glassfish.java.net/Wiki.jsp?page=TopLinkEssentials>  
8 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FPersistence%20Products>  
9 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>



## 14 Kodo

Kodo<sup>1</sup>, was originally developed as an Java Data Objects (JDO) implementation, by SolarMetric. BEA Systems acquired SolarMetric in 2005, where Kodo was expanded to be an implementation of both the JDO and JPA specifications. In 2006, BEA donated a large part of the Kodo source code to the Apache Software Foundation under the name OpenJPA. BEA (and Kodo) were acquired by Oracle. Kodo development is now stopped and users are referred to EclipseLink

Kodo<sup>2</sup> Kodo<sup>3</sup>

---

<sup>1</sup> <http://www.bea.com/kodo/>

<sup>2</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FPersistence%20Products>

<sup>3</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>



## 15 Open JPA

OpenJPA<sup>1</sup> is an Apache project for supporting the JPA specification. Its source code was originally donated from (part of) BEA's Kodo product.

Open JPA<sup>2</sup> Open JPA<sup>3</sup>

---

<sup>1</sup> <http://en.wikipedia.org/wiki/OpenJPA>

<sup>2</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FPersistence%20Products>

<sup>3</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>





# 16 Ebean

Ebean<sup>1</sup> is a Java ORM based on the JPA specification.

The goal of Ebean is to provide mapping<sup>2</sup> compatible with JPA while providing a simpler API to use and learn.

## 16.1 Index

1. /Why Ebean/<sup>3</sup>
2. /Example Model/<sup>4</sup>
3. /Quick Start - examples/<sup>5</sup>
4. /Mapping via JPA Annotations/<sup>6</sup>
5. /Query/<sup>7</sup>
  - a) /Basics/<sup>8</sup>
  - b) /Future - Asynchronous query execution/<sup>9</sup>
  - c) /PagingList/<sup>10</sup>
  - d) /Aggregation - Group By/<sup>11</sup>
6. /Save and Delete/<sup>12</sup>
7. /Transactions/<sup>13</sup>
8. /Caching/<sup>14</sup>
9. /Using raw SQL/<sup>15</sup>

---

1 <http://www.avaje.org>  
2 Chapter 17 on page 37  
3 <http://en.wikibooks.org/wiki/%2FWhy%20Ebean%2F>  
4 <http://en.wikibooks.org/wiki/%2FExample%20Model%2F>  
5 <http://en.wikibooks.org/wiki/%2FQuick%20Start%20-%20examples%2F>  
6 <http://en.wikibooks.org/wiki/%2FMapping%20via%20JPA%20Annotations%2F>  
7 <http://en.wikibooks.org/wiki/%2FQuery%2F>  
8 <http://en.wikibooks.org/wiki/%2FBasics%2F>  
9 <http://en.wikibooks.org/wiki/%2FFuture%20-%20Asynchronous%20query%20execution%2F>  
10 <http://en.wikibooks.org/wiki/%2FPagingList%2F>  
11 <http://en.wikibooks.org/wiki/%2FAggregation%20-%20Group%20By%2F>  
12 <http://en.wikibooks.org/wiki/%2FSave%20and%20Delete%2F>  
13 <http://en.wikibooks.org/wiki/%2FTransactions%2F>  
14 <http://en.wikibooks.org/wiki/%2FCaching%2F>  
15 <http://en.wikibooks.org/wiki/%2FUsing%20raw%20SQL%2F>



## 17 Mapping



# 18 Mapping

The first thing that you need to do to persist something in Java is define how it is to be persisted. This is called the mapping process ( details<sup>1</sup>). There have been many different solutions to the mapping process over the years, including some object databases that didn't require you map anything but let you persist anything directly. Object-relational mapping tools that would generate an object model for a data model that included the mapping and persistence logic in it. ORM products that provided mapping tools to allow the mapping of an existing object model to an existing data model and stored this mapping meta-data in flat files, database tables, XML and finally annotations.

In JPA mappings can either be stored through Java annotations, or in XML files. One significant aspect of JPA is that only the minimal amount of mapping is required. JPA implementations are required to provide defaults for almost all aspects of mapping an object.

The minimum requirement to mapping an object in JPA is to define which objects can be persisted. This is done through either marking the class with the `@Entity`<sup>2</sup> annotation, or adding an `<entity>` tag for the class in the persistence unit's ORM XML file. Also the primary key, or unique identifier attribute(s) must be defined for the class. This is done through marking one of the class' fields or properties (get method) with the `@Id` annotation, or adding an `<id>` tag for the class' attribute in the ORM XML<sup>3</sup> file.

The JPA implementation will default all other mapping information, including defaulting the table name, column names for all defined fields or properties, cardinality and mapping of relationships, all SQL and persistence logic for accessing the objects. Most JPA implementations also provide the option of generating the database tables at runtime, so very little work is required by the developer to rapidly develop a persistent JPA application.

---

1 [http://en.wikipedia.org/wiki/Object-Relational\\_impedance\\_mismatch](http://en.wikipedia.org/wiki/Object-Relational_impedance_mismatch)

2 <https://java.sun.com/javase/5/docs/api/javax/persistence/Entity.html>

3 [http://java.sun.com/xml/ns/persistence/orm\\_1\\_0.xsd](http://java.sun.com/xml/ns/persistence/orm_1_0.xsd)

## 18.0.1 Example object model

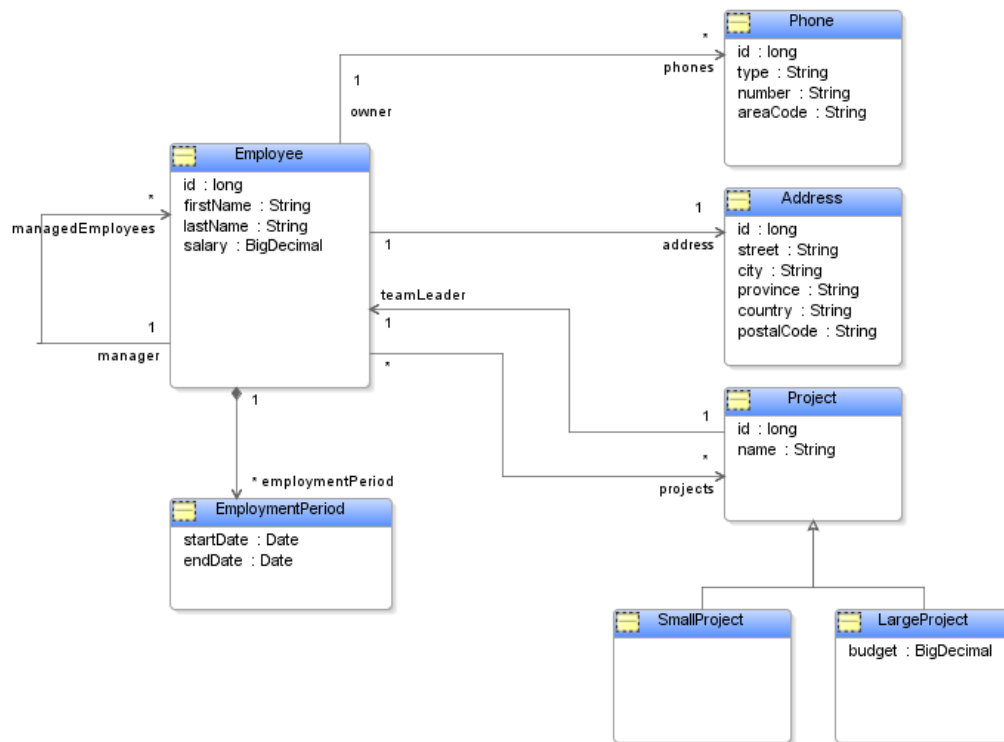


Figure 5

## 18.0.2 Example data model

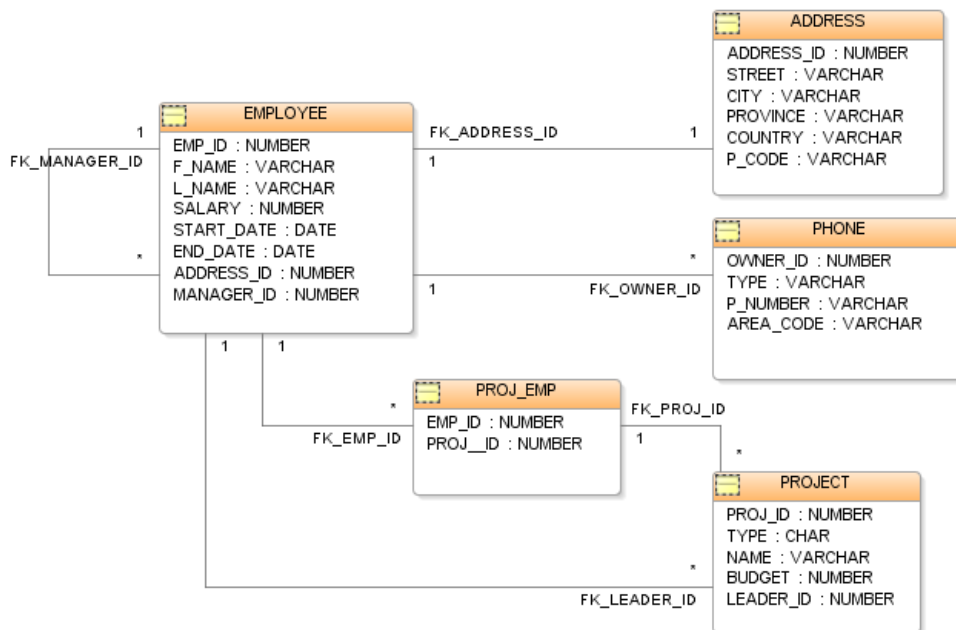


Figure 6

## 18.0.3 Example of a persistent entity mapping in annotations

```

import javax.persistence.*;
...
@Entity
public class Employee {
    @Id
    private long id;
    private String firstName;
    private String lastName;
    private Address address;
    private List<Phone> phones;
    private Employee manager;
    private List<Employee> managedEmployees;
    ...
}

```

## 18.0.4 Example of a persistent entity mapping in XML

```

<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings version="1.0"
  xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

```



```
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
orm_1_0.xsd">
  <description>The minimal mappings for a persistent entity in
XML.</description>
  <entity name="Employee" class="org.acme.Employee" access="FIELD">
    <attributes>
      <id name="id"/>
    </attributes>
  </entity>
</entity-mappings>
```

## 18.1 Access Type

JPA allows annotations to be placed on either the class field, or on the `get` method for the property. This defines the *access type* of the class, which is either `FIELD` or `PROPERTY`. Either all annotations must be on the fields, or all annotations on the `get` methods, but not both (unless the `@AccessType` annotation is used). JPA does not define a default access type (oddly enough), so the default may depend on the JPA provider. The default is assumed to occur based on where the `@Id` annotation is placed, if placed on a field, then the class is using `FIELD` access, if placed on a `get` method, then the class is using `PROPERTY` access. The access type can also be defined through XML on the `<entity>` element. The access type can be configured using the `@AccessType`<sup>4</sup> annotation or `access-type` XML attribute.

For `FIELD` access the class field value will be accessed directly to store and load the value from the database. This is normally done either through reflection, or through generated byte code, but depends on the JPA provider and configuration. The field can be `private` or any other access type. `FIELD` is normally safer, as it avoids any unwanted side-affect code that may occur in the application `get/set` methods.

For `PROPERTY` access the class `get` and `set` methods will be used to store and load the value from the database. This is normally done either through reflection, or through generated byte code, but depends on the JPA provider and configuration. `PROPERTY` has the advantage of allowing the application to perform conversion of the database value when storing it in the object. The user should be careful to not put any side-affects in the `get/set` methods that could interfere with persistence.

JPA 2.0 allows the access type to also be set on a specific field or `get` method. This allows for the class to use one default access mechanism, but for one attribute to use a different access type. This can be used for attributes that need to be converted through a set of database specific `get`, `set`, or allow a specific attribute to avoid side-affects in its `get`, `set` methods. This is done through specifying the `@AccessType` annotation or XML attribute on the mapped attribute. You may also need to mark the field/property as `@Transient` if it has a different attribute name than the mapped attribute.

JPA allows for a persistence unit default `<access-type>` element to be set in `persistence-unit-defaults` or entity default in `entity-mappings`.

---

4 <https://java.sun.com/javaee/5/docs/api/javax/persistence/AccessType.html>

TopLink<sup>5</sup> / EclipseLink<sup>6</sup> : Default to using FIELD access. If weaving is enabled, and field access is used the fields are accessed directly using generated byte-code. If not using weaving, or using property access, reflection is used. EclipseLink also supports a third access type VIRTUAL, which can be used from the ORM XML to map dynamic properties stored in a properties Map.

### 18.1.1 Access type example

```
@Entity
@Access(AccessType.FIELD)
public class Employee {
    @Id
    private long id;
    private String firstName;
    private String lastName;
    @Transient
    private Money salary;
    @ManyToOne(fetch=FetchType.LAZY)
    private Employee manager;

    @Access(AccessType.PROPERTY)
    private BigDecimal getBDSalary() {
        return this.salary.toNumber();
    }
    private void setBDSalary(BigDecimal salary) {
        this.salary = new Money(salary);
    }

    private Money getSalary() {
        return this.salary;
    }
    private void setSalary(Money salary) {
        this.salary = salary;
    }
    ...
}
```

## 18.2 Common Problems

### *My annotations are ignored*

This typically occurs when you annotate both the fields and methods (properties) of the class. You must choose either field or property access, and be consistent. Also when annotating properties you must put the annotation on the get method, not the set method. Also ensure that you have not defined the same mappings in XML, which may be overriding the annotations. You may also have a classpath issue, such as having an old version of the class on the classpath.

---

5 Chapter 11 on page 25

6 Chapter 10 on page 23

### *Odd behavior*

There are many reasons that odd behavior can occur with persistence. One common issue that can cause odd behavior is using property access and putting *side effects* in your get or set methods. For this reason it is generally recommended to use field access in mapping, i.e. putting your annotations on your variables not your get methods.

For example consider:

```
public void setPhones(List<Phone> phones) {  
    for (Phone phone : phones) {  
        phone.setOwner(this);  
    }  
    this.phones = phones;  
}
```

This may look innocent, but these side effects can have unexpected consequences. For example if the relationship was **lazy** this would have the effect of always instantiating the collection when set from the database. It could also have consequences with certain JPA implementations for persisting, merging and other operations, causing duplicate inserts, missed updates, or a corrupt object model.

I have also seen simply incorrect property methods, such as a get method that always returns a new object, or a copy, or set methods that don't actually set the value.

In general if you are going to use property access, ensure your property methods are free of side effects. Perhaps even use different property methods than your application uses.

Mapping<sup>7</sup> Mapping<sup>8</sup>

---

<sup>7</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

<sup>8</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FMapping>

## 19 Tables

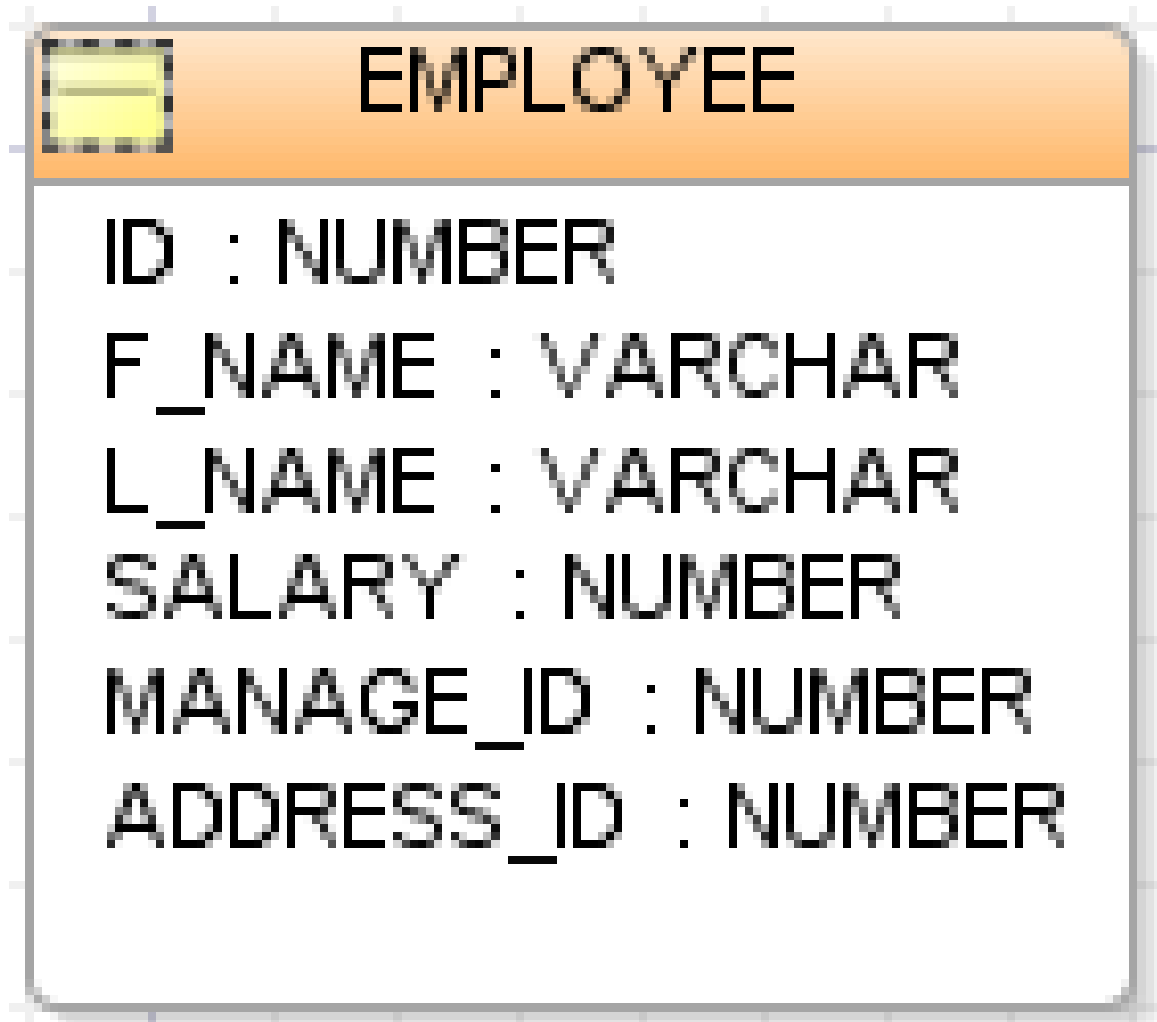


Figure 7

A table<sup>1</sup> is the basic persist structure of a relational database. A table contains a list of columns which define the table's structure, and a list of rows that define the table's data. Each column has a specific type and generally size. The standard set of relational types are limited to basic types including numeric, character, date-time, and binary (although most modern databases have additional types and typing systems). Tables can also have constraints that define the rules which restrict the row data, such as primary key, foreign

<sup>1</sup> <http://en.wikipedia.org/wiki/Table%20%28database%29>

key, and unique constraints. Tables also have other artifacts such as indexes, partitions and triggers.

A typical mapping of a persist class will map the class to a single table. In JPA this is defined through the `@Table2` annotation or `<table>` XML element. If no table annotation is present, the JPA implementation will auto assign a table for the class. The JPA default table name is the name of the class (minus the package) with the first letter capitalized. Each attribute of the class will be stored in a column in the table.

### 19.0.1 Example mapping annotations for an entity with a single table

```
...
@Entity
@Table(name="EMPLOYEE")
public class Employee {
    ...
}
```

### 19.0.2 Example mapping XML for an entity with a single table

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
    <table name="EMPLOYEE"/>
</entity>
```

---

2 <https://java.sun.com/javase/5/docs/api/javax/persistence/Table.html>

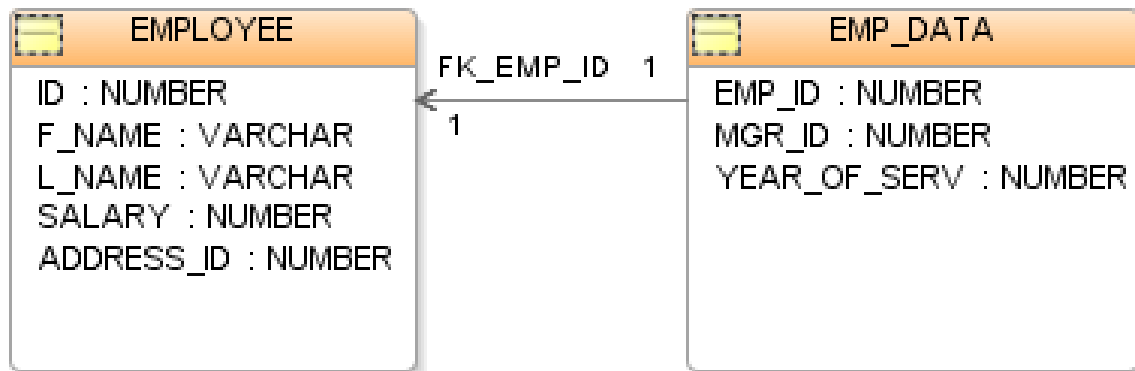
## 20 Advanced

Although in the ideal case each class would map to a single table, this is not always possible. Other scenarios include:

- Multiple tables<sup>1</sup> : One class maps to 2 or multiple tables.
- Sharing tables<sup>2</sup> : 2 or multiple classes are stored in the same table.
- Inheritance<sup>3</sup> : A class is involved in inheritance and has an inherited and local table.
- Views<sup>4</sup> : A class maps to a view.
- Stored procedures<sup>5</sup> : A class maps to a set of stored procedures.
- Partitioning<sup>6</sup> : Some instances of a class map to one table, and other instances to another table.
- Replication<sup>7</sup> : A class's data is replicated to multiple tables.
- History<sup>8</sup> : A class has historical data.

These are all advanced cases, some are handled by the JPA Spec and many are not. The following sections investigate each of these scenarios further and include what is supported by the JPA spec, what can be done to workaround the issue within the spec, and how to use some JPA implementations extensions to handle the scenario.

### 20.1 Multiple tables



**Figure 8**

- 
- 1 Chapter 20.1 on page 47
  - 2 Chapter 41.7 on page 201
  - 3 Chapter 23.6 on page 75
  - 4 Chapter 41.2 on page 196
  - 5 Chapter 41.4 on page 198
  - 6 Chapter 41.12 on page 204
  - 7 Chapter 41.11 on page 203
  - 8 Chapter 41.8 on page 202

Sometimes a class maps to multiple tables. This typically occurs on legacy or existing data models where the object model and data model do not match. It can also occur in inheritance when subclass data is stored in additional tables. Multiple tables may also be used for performance, partitioning or security reasons.

JPA allows multiple tables to be assigned to a single class. The `@SecondaryTable`<sup>9</sup> and `SecondaryTables` annotations or `<secondary-table>` elements can be used. By default the `@Id` column(s) are assumed to be in both tables, such that the secondary table's `@Id` column(s) are the primary key of the secondary table and a foreign key to the first table. If the first table's `@Id` column(s) are not named the same the `@PrimaryKeyJoinColumn`<sup>10</sup> or `<primary-key-join-column>` can be used to define the foreign key join condition.

In a multiple table entity, each mapping must define which table the mapping's columns are from. This is done using the `table` attribute of the `@Column` or `@JoinColumn` annotations or XML elements. By default the primary table of the class is used, so you only need to set the table for secondary tables. For inheritance the default table is the primary table of the subclass being mapped.

### 20.1.1 Example mapping annotations for an entity with multiple tables

```
...
@Entity
@Table(name="EMPLOYEE")
@SecondaryTable(name="EMP_DATA",
    pkJoinColumns = @PrimaryKeyJoinColumn(name="EMP_ID",
        referencedColumnName="ID")
    )
public class Employee {
    ...
    @Column(name="YEAR_OF_SERV", table="EMP_DATA")
    private int yearsOfService;

    @OneToOne
    @JoinColumn(name="MGR_ID", table="EMP_DATA",
        referencedColumnName="ID")
    private Employee manager;
    ...
}
```

### 20.1.2 Example mapping XML for an entity with multiple tables

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <table name="EMPLOYEE"/>
  <secondary-table name="EMP_DATA">
    <primary-key-join-column name="EMP_ID"
      referenced-column-name="ID"/>
  </secondary-table>
  <attributes>
    ...
    <basic name="yearsOfService">
      <column name="YEAR_OF_SERV" table="EMP_DATA"/>
    </basic>
  </attributes>
</entity>
```

---

9 <https://java.sun.com/javaee/5/docs/api/javax/persistence/SecondaryTable.html>

10 <https://java.sun.com/javaee/5/docs/api/javax/persistence/PrimaryKeyJoinColumn.html>

```

    <one-to-one name="manager">
      <join-column name="MGR_ID" table="EMP_DATA"
referenced-column-name="ID"/>
    </one-to-one>
  </attributes>
</entity>

```

With the `@PrimaryKeyJoinColumn` the name refers to the foreign key column in the secondary table and the `referencedColumnName` refers to the primary key column in the first table. If you have multiple secondary tables, they must always refer to the first table. When defining the table's schema typically you will define the join columns in the secondary table as the primary key of the table, and a foreign key to the first table. Depending how you have defined your foreign key constraints, the order of the tables can be important, the order will typically match the order that the JPA implementation will insert into the tables, so ensure the table order matches your constraint dependencies.

For relationships to a class that has multiple tables the foreign key (join column) always maps to the primary table of the target. JPA does not allow having a foreign key map to a table other than the target object's primary table. Normally this is not an issue as foreign keys almost always map to the id/primary key of the primary table, but in some advanced scenarios this may be an issue. Some JPA products allow the column or join column to use the qualified name of the column (i.e. `@JoinColumn(referenceColumnName="EMP_DATA.EMP_NUM")`), to allow this type of relationship. Some JPA products may also support this through their own API, annotations or XML.

## 20.2 Multiple tables with foreign keys

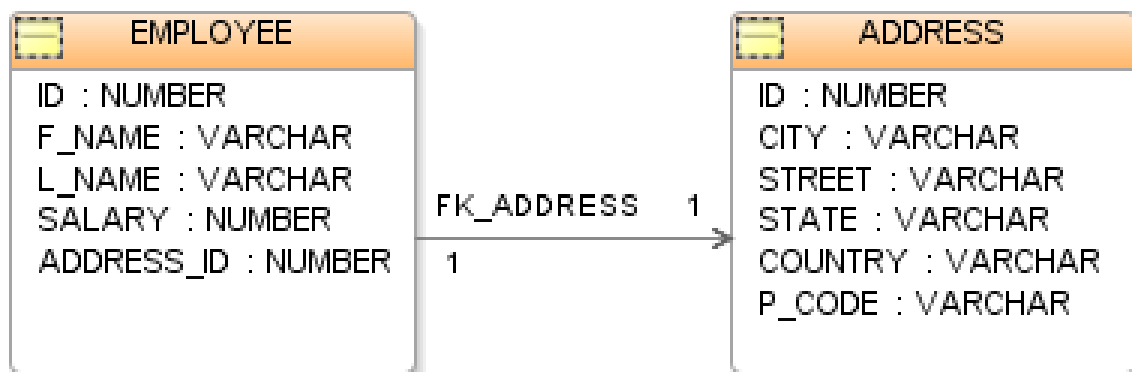


Figure 9

Sometimes you may have a secondary table that is referenced through a foreign key from the primary table to the secondary table instead of a foreign key from the secondary table to the primary table. You may even have a foreign key between two of the secondary tables. Consider having an `EMPLOYEE` and `ADDRESS` table where `EMPLOYEE` refers to `ADDRESS` through an `ADDRESS_ID` foreign key, and (for some strange reason) you only want a single `Employee` class that has the data from both tables. The JPA spec does not cover this directly, so if you have this scenario the first thing to consider, if you have the flexibility, is to change your data model to stay within the confines of the spec. You could also change your object



model to define a class for each table, in this case an `Employee` class and an `Address` class, which is typically the best solution. You should also check with your JPA implementation to see what extensions it supports in this area.

One way to solve the issue is simply to swap your primary and secondary tables. This will result in having the secondary table referencing the primary table's primary key and is within the spec. This however will have side-effects, one being that you now changed the primary key of your object from `EMP_ID` to `ADDRESS_ID`, and may have other mapping and querying implications. If you have more than 2 tables this also may not work.

Another option is to just use the foreign key column in the `@PrimaryKeyJoinColumn`, this will technically be backward, and perhaps not supported by the spec, but may work for some JPA implementations. However this will result in the table insert order not matching the foreign key constraints, so the constraints will need to be removed, or deferred.

It is also possible to map the scenario through a database view. A view could be defined joining the two tables and the class could be mapped to the view instead of the tables. Views are read-only on some databases, but many also allow writes, or allow triggers to be used to handle writes.

Some JPA implementations provide extensions to handle this scenario.

TopLink<sup>11</sup>, EclipseLink<sup>12</sup> : Provides a proprietary API for its mapping model `ClassDescriptor.addForeignKeyFieldNameForMultipleTable()` that allows for arbitrary complex foreign key relationships to be defined among the secondary tables. This can be configured through using a `@DescriptorCustomizer` annotation and `DescriptorCustomizer` class.

## 20.3 Multiple table joins

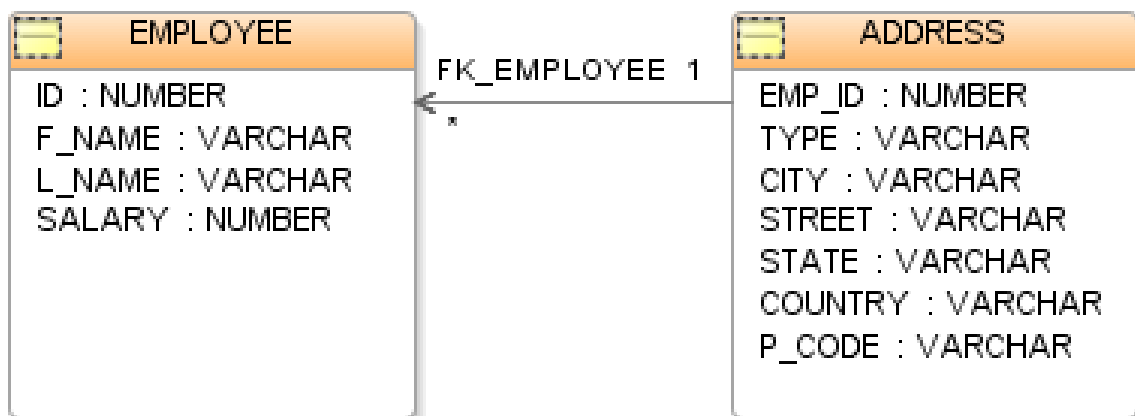


Figure 10

<sup>11</sup> Chapter 11 on page 25

<sup>12</sup> Chapter 10 on page 23

Occasionally the data model and object model do not get along very well at all. The database could be a legacy model and not fit very well with the new application model, or the DBA or object architect may be a little crazy. In these cases you may require advanced multiple table joins.

Examples of these include having two tables related not by their primary or foreign keys, but through some constant or computation. Consider having an **EMPLOYEE** table and an **ADDRESS** table, the **ADDRESS** table has an **EMP\_ID** foreign key to the **EMPLOYEE** table, but there are several address for each employee and only the address with the **TYPE** of "HOME" is desired. In this case data from both of the tables is desired to be mapped in the **Employee** object. A join expression is required where the foreign key matches and the constant matches.

Again this scenario could be handled through redesigning the data or object model, or through using a view. Some JPA implementations provide extensions to handle this scenarios.

TopLink<sup>13</sup>, EclipseLink<sup>14</sup> : Provides a proprietary API for its mapping model `DescriptorQueryManager.setMultipleTableJoinExpression()` that allows for arbitrary complex multiple table joins to be defined. This can be configured through using a `@DescriptorCustomizer` annotation and `DescriptorCustomizer` class.

## 20.4 Multiple table outer joins

Another perversion of multiple table mapping is to desire to outer join the secondary table. This may be desired if the secondary may or may not have a row defined for the object. Typically the object should be read-only if this is to be attempted, as writing to a row that may or may not be there can be tricky.

This is not directly supported by JPA, and it is best to reconsider the data model or object model design if faced with this scenario. Again it is possible to map this through a database view, where an outer join is used to join the tables in the view.

Some JPA implementation support using outer joins for multiple tables.

Hibernate<sup>15</sup> : This can be accomplished through using the Hibernate `@Table` annotation and set its `optional` attribute to `true`. This will configure Hibernate to use an outer join to read the table, and will not write to the table if all of the attributes mapping to the table are null.

TopLink<sup>16</sup>, EclipseLink<sup>17</sup> : If the database supports usage of outer join syntax in the where clause (Oracle, Sybase, SQL Server), then the multiple table join expression could be used to configure an outer join to be used to read the table.

---

13 Chapter 11 on page 25

14 Chapter 10 on page 23

15 Chapter 12 on page 27

16 Chapter 11 on page 25

17 Chapter 10 on page 23

## 20.5 Tables with special characters and mixed case

Some JPA providers may have issues with table and column names with special characters, such as spaces. In general it is best to use standard characters, no spaces, and all uppercase names. International languages should be ok, as long as the database and JDBC driver supports the character set.

It may be required to "quote" table and column names with special characters or in some cases with mixed case. For example if the table name had a space it could be defined as the following:

```
@Table("\"Employee Data\"")
```

Some databases support mixed case table and column names, and others are case insensitive. If your database is case insensitive, or you wish your data model to be portable, it is best to use all uppercase names. This is normally not a big deal with JPA where you rarely use the table and column names directly from your application, but can be an issue in certain cases if using native SQL queries.

## 20.6 Table qualifiers, schemas, or creators

A database table may require to be prefixed with a table qualifier, such as the table's creator, or its' namespace, schema, or catalog. Some databases also support linking table on other database, so the link name can also be a table qualifier.

In JPA a table qualifier can be set on a table through the `schema` or `catalog` attribute. Generally it does not matter which attribute is used as both just result in prefixing the table name. Technically you could even include the full name "schema.table" as the table's name and it would work. The benefit of setting the prefix in the schema or catalog is a *default* table qualifier can be set for the entire persistence unit, also not setting the real table name may impact native SQL queries.

If all of your tables require the same table qualifier, you can set the default in the `orm.xml`.

### 20.6.1 Example mapping annotations for an entity with a qualified table

```
...
@Entity
@Table(name="EMPLOYEE", schema="ACME")
public class Employee {
    ...
}
```

### 20.6.2 Example mapping XML for default (entire persistence unit) table qualifier

```
<entity-mappings>
  <persistence-unit-metadata>
```

```
    <persistence-unit-defaults>
      <schema name="ACME"/>
    </persistence-unit-defaults>
  </persistence-unit-metadata>
  ....
</entity-mappings>
```

### 20.6.3 Example mapping XML for default (orm file) table qualifier

```
<entity-mappings>
  <schema name="ACME"/>
  ...
</entity-mappings>
```

Tables <sup>18</sup> Tables <sup>19</sup>

---

<sup>18</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

<sup>19</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FMapping>



## 21 Identity

An object id (OID) is something that uniquely identifies an object. Within a VM this is typically the object's pointer. In a relational database table a row is uniquely identified in its table by its primary key<sup>1</sup>. When persisting objects to a database you need a unique identifier for the objects, this allows you to query the object, define relationships to the object, and update and delete the object. In JPA the object id is defined through the `@Id`<sup>2</sup> annotation or `<id>` element and should correspond to the primary key of the object's table.

### 21.0.4 Example id annotation

```
...
@Entity
public class Employee {
    @Id
    private long id
    ...
}
```

### 21.0.5 Example id XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
  </attributes>
</entity>
```

### Common Problems

*Strange behavior, unique constraint violation.*

You must never change the id of an object. Doing so will cause errors, or strange behavior depending on your JPA provider. Also do not create two objects with the same id, or try persisting an object with the same id as an existing object. If you have an object that may be existing use the `EntityManager merge()` API, do not use `persist()` for an existing object, and avoid relating an un-managed existing object to other managed objects.

---

1 <http://en.wikipedia.org/wiki/Primary%20key>

2 <https://java.sun.com/javase/5/docs/api/javax/persistence/Id.html>

*No primary key.*

See No Primary Key<sup>3</sup>.

---

<sup>3</sup> Chapter 23.6 on page 75

## 22 Sequencing

An object id can either be a natural id or a generated id. A natural id is one that occurs in the object and has some meaning in the application. Examples of natural ids include user ids, email addresses, phone numbers, and social insurance numbers. A generated id is one that is generated by the system. A sequence number in JPA is a sequential id generated by the JPA implementation and automatically assigned to new objects. The benefits of using sequence numbers are that they are guaranteed to be unique, allow all other data of the object to change, are efficient values for querying and indexes, and can be efficiently assigned. The main issue with natural ids is that everything always changes at some point; even a person's social insurance number can change. Natural ids can also make querying, foreign keys and indexing less efficient in the database.

In JPA an `@Id` can be easily assigned a generated sequence number through the `@GeneratedValue`<sup>1</sup> annotation, or `<generated-value>` element.

### 22.0.6 Example generated id annotation

```
...
@Entity
public class Employee {
    @Id
    @GeneratedValue
    private long id
    ...
}
```

### 22.0.7 Example generated id XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id">
      <generated-value/>
    </id>
  </attributes>
</entity>
```

## 22.1 Sequence Strategies

There are several strategies for generating unique ids. Some strategies are database agnostic and others make use of built-in databases support.

---

1 <https://java.sun.com/javaee/5/docs/api/javax/persistence/GeneratedValue.html>



JPA provides support for several strategies for id generation defined through the `GenerationType`<sup>2</sup> enum values: `TABLE`, `SEQUENCE` and `IDENTITY`.

The choice of which sequence strategy to use is important as it affects performance, concurrency and portability.

### 22.1.1 Table sequencing

Table sequencing uses a table in the database to generate unique ids. The table has two columns, one stores the name of the sequence, the other stores the last id value that was assigned. There is a row in the sequence table for each sequence object. Each time a new id is required the row for that sequence is incremented and the new id value is passed back to the application to be assigned to an object. This is just one example of a sequence table schema, for other table sequencing schemas see Customizing<sup>3</sup>.

Table sequencing is the most portable solution because it just uses a regular database table, so unlike sequence and identity can be used on any database. Table sequencing also provides good performance because it allows for sequence pre-allocation, which is extremely important to insert performance, but can have potential concurrency issues<sup>4</sup>.

In JPA the `@TableGenerator`<sup>5</sup> annotation or `<table-generator>` element is used to define a sequence table. The `TableGenerator` defines a `pkColumnName` for the column used to store the name of the sequence, `valueColumnName` for the column used to store the last id allocated, and `pkColumnValue` for the value to store in the name column (normally the sequence name).

#### Example sequence table

SEQUENCE\_TABLE

SEQ_NAME	SEQ_COUNT
EMP_SEQ	123
PROJ_SEQ	550

#### Example table generator annotation

```
...
@Entity
public class Employee {
    @Id
    @TableGenerator(name="TABLE_GEN", table="SEQUENCE_TABLE",
pkColumnName="SEQ_NAME",
valueColumnName="SEQ_COUNT", pkColumnValue="EMP_SEQ")
    @GeneratedValue(strategy=GenerationType.TABLE,
```

---

2 <https://java.sun.com/javaee/5/docs/api/javax/persistence/GenerationType.html>

3 Chapter 23.3.2 on page 74

4 Chapter 23.3 on page 73

5 <https://java.sun.com/javaee/5/docs/api/javax/persistence/TableGenerator.html>

```

generator="TABLE_GEN")
    private long id;
    ...
}

```

## Example table generator XML

```

<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id">
      <generated-value strategy="TABLE" generator="EMP_SEQ"/>
      <table-generator name="EMP_SEQ" table="SEQUENCE_TABLE"
pk-column-name="SEQ_NAME"
      value-column-name="SEQ_COUNT"
pk-column-value="EMP_SEQ"/>
    </id>
  </attributes>
</entity>

```

## Common Problems

### *Error when allocating a sequence number.*

Errors such as *"table not found"*, *"invalid column"* can occur if you do not have a SEQUENCE table defined in your database, or its schema does not match what you have configured, or what your JPA provider is expecting by default. Ensure you create the sequence table correctly, or configure your `@TableGenerator` to match the table that you created, or let your JPA provider create you tables for you (most JPA provider support schema creation). You may also get an error such as *"sequence not found"*, this means you did not create a row in the table for your sequence. You must insert an initial row in the sequence table for your sequence with the initial id (i.e. `INSERT INTO SEQUENCE_TABLE (SEQ_NAME, SEQ_COUNT) VALUES ("EMP_SEQ", 0)`), or let your JPA provider create your schema for you.

### *Deadlock or poor concurrency in the sequence table.*

See concurrency issues<sup>6</sup>.

## 22.1.2 Sequence objects

Sequence objects use special database objects to generate ids. Sequence objects are only supported in some databases, such as Oracle, DB2, and Postgres. Usually, a SEQUENCE object has a name, an INCREMENT, and other database object settings. Each time the `<sequence>.NEXTVAL` is selected the sequence is incremented by the INCREMENT.

Sequence objects provide the optimal sequencing option, as they are the most efficient and have the best concurrency, however they are the least portable as most databases do

---

<sup>6</sup> Chapter 23.3 on page 73

not support them. Sequence objects support sequence preallocation through setting the INCREMENT on the database sequence object to the sequence preallocation size.

In JPA the `@SequenceGenerator`<sup>7</sup> annotation or `<sequence-generator>` element is used to define a sequence object. The `SequenceGenerator` defines a `sequenceName` for the name of the database sequence object, and an `allocationSize` for the sequence preallocation size or sequence object INCREMENT.

### Example sequence generator annotation

```
...
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE,
generator="EMP_SEQ")
    @SequenceGenerator(name="EMP_SEQ", sequenceName="EMP_SEQ",
allocationSize=100)
    private long id;
    ...
}
```

### Example sequence generator XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id">
      <generated-value strategy="SEQUENCE"
generator="EMP_SEQ"/>
      <sequence-generator name="EMP_SEQ"
sequence-name="EMP_SEQ" allocation-size="100"/>
    </id>
  </attributes>
</entity>
```

## Common Problems

### *Error when allocating a sequence number.*

Errors such as *"sequence not found"*, can occur if you do not have a SEQUENCE object defined in your database. Ensure you create the sequence object, or let your JPA provider create your schema for you (most JPA providers support schema creation). When creating your sequence object, ensure the sequence's INCREMENT matches your `SequenceGenerator`'s `allocationSize`. The DDL to create a sequence object depends on the database, for Oracle it is, `CREATE SEQUENCE EMP_SEQ INCREMENT BY 100 START WITH 100`.

---

<sup>7</sup> <https://java.sun.com/javase/5/docs/api/javax/persistence/SequenceGenerator.html>

***Invalid, duplicate or negative sequence numbers.***

This can occur if your sequence object's INCREMENT does not match your `allocationSize`. This results in the JPA provider thinking it got back more sequences than it really did, and ends up duplicating values, or with negative numbers. This can also occur on some JPA providers if your sequence object's STARTS WITH is 0 instead of a value equal or greater to the `allocationSize`.

**22.1.3 Identity sequencing**

Identity sequencing uses special IDENTITY columns in the database to allow the database to automatically assign an id to the object when its row is inserted. Identity columns are supported in many databases, such as MySQL, DB2, SQL Server, Sybase, and PostgreSQL. Oracle does not support IDENTITY columns but it is possible to simulate them using sequence objects and triggers.

Although identity sequencing seems like the easiest method to assign an id, they have several issues. One is that since the id is not assigned by the database until the row is inserted the id cannot be obtained in the object until after commit or after a flush call. Identity sequencing also does not allow for sequence preallocation, so can require a select for each object that is inserted, potentially causing a major performance problem, so in general are not recommended.

In JPA there is no annotation or element for identity sequencing as there is no additional information to specify. Only the `GeneratedValue`'s strategy needs to be set to IDENTITY.

**Example identity annotation**

```
...
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;
    ...
}
```

**Example identity XML**

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id">
      <generated-value strategy="IDENTITY"/>
    </id>
  </attributes>
</entity/>
```

## PostgreSQL Serial Columns

Sequence objects are database constructs can be instantiated as distinct objects as needed. They provide incremental values when their "next" method is called. Note that the increments can be values greater than one, for example incremented by two's, three's etc. An 'on insert' trigger can be constructed for each table that calls a sequence object created for that specific table primary key asking for the next value and inserting it into the table with the new record. Identity datatypes in products like MySQL or SQL Server are encapsulated types that do the same thing without requiring the set up of triggers and sequences (although PostgreSQL at least has automated much of this with its Serial and Big Serial *pseudo* datatypes (these actually create an int/bigint column and run a "macro" creating the sequence and 'on insert' trigger when the CREATE TABLE statement is executed.)).

## Common Problems

*null is inserted into the database, or error on insert.*

This typically occurs because the @Id was not configured to use an @GeneratedValue(strategy=GenerationType.IDENTITY). Ensure it is configured correctly. It could also be that your JPA provider does not support identity sequencing on the database platform that you are using, or you have not configured your database platform. Most providers require that you set the database platform through a persistence.xml property, most provider also allow you to customize your own platform if it is not directly supported. It may also be that you did not set your primary key column in your table to be an identity type.

*Object's id is not assigned after persist.*

Identity sequencing requires the insert to occur before the id can be assigned, so it is not assigned on persist like other types of sequencing. You must either call `commit()` on the current transaction, or call `flush()` on the `EntityManager`. It may also be that you did not set your primary key column in your table to be an identity type.

*Child's id is not assigned from parent on persist.*

A common issue is that the generated Id is part of a child object's Id through a `OneToOne` or `ManyToOne` mapping. In this case, because JPA requires that the child define a duplicate `Basic` mapping for the Id, its Id will be inserted as null. One solution to this is to mark the `Column` on the Id mapping in the child as `insertable=false`, `updateable=false`, and define the `OneToOne` or `ManyToOne` using a normal `JoinColumn` this will ensure the foreign key field is populated by the `OneToOne` or `ManyToOne` not the `Basic`. Another option is to first persist the parent, then call `flush()` before persisting the child.

***Poor insert performance.***

Identity sequencing does not support sequence preallocation, so requires a select after each insert, in some cases doubling the insert cost. Consider using a sequence table, or sequence object to allow sequence preallocation.

***Lost latest free id.***

MySQL bug 199<sup>8</sup> causes its autoincrement counter to be lost on restart. So if the last entity is removed and the MySQL server restarted, the same id will be re-used and thus not be unique.

---

<sup>8</sup> <http://bugs.mysql.com/bug.php?id=199>



## 23 Advanced

### 23.1 Composite Primary Keys

A composite primary key is one that is made up of several columns in the table. A composite primary key can be used if no single column in the table is unique. It is generally more efficient and simpler to have a one-column primary key, such as a generated sequence number, but sometimes a composite primary key is desirable and unavoidable.

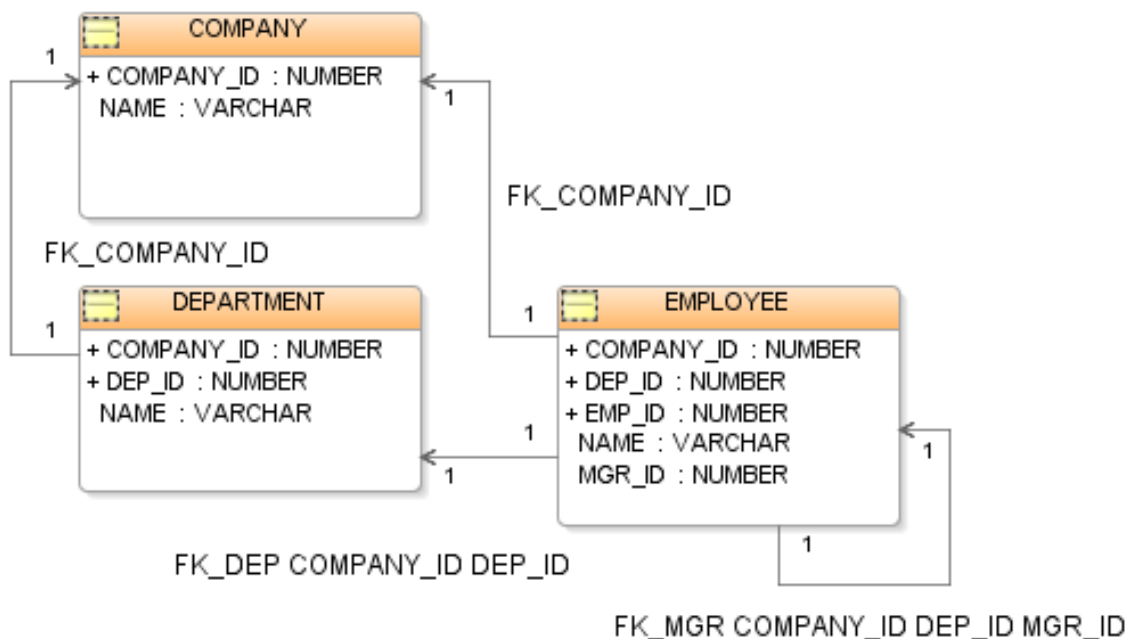


Figure 11

Composite primary keys are common in legacy database schemas, where *cascaded keys* can sometimes be used. This refers to a model where dependent objects' key definitions include their parents' primary key; for example, `COMPANY`'s primary key is `COMPANY_ID`, `DEPARTMENT`'s primary key is composed of a `COMPANY_ID` and a `DEP_ID`, `EMPLOYEE`'s primary key is composed of `COMPANY_ID`, `DEP_ID`, and `EMP_ID`, and so on. Although this generally does not match object-oriented design principles, some DBA's prefer this model. Difficulties with the model include the restriction that employees cannot switch departments, that foreign-key relationships become more complex, and that all primary-key operations (including queries, updates, and deletes) are less efficient. However, each department has control over its own employee IDs, and if needed the database `EMPLOYEE` table can be partitioned based on the `COMPANY_ID` or `DEP_ID`, as these are included in every query.



Other common usages of composite primary keys include many-to-many relationships where the join table has additional columns, so the table itself is mapped to an object whose primary key consists of the pair of foreign-key columns and dependent or aggregate one-to-many relationships where the child object's primary key consists of its parent's primary key and a locally unique field.

There are two methods of declaring a composite primary key in JPA, `IdClass` and `EmbeddedId`.

### 23.1.1 Id Class

An `IdClass` defines a separate Java class to represent the primary key. It is defined through the `@IdClass`<sup>1</sup> annotation or `<id-class>` XML element. The `IdClass` must define an attribute (field/property) that mirrors each `Id` attribute in the entity. It must have the same attribute name and type. When using an `IdClass` you still require to mark each `Id` attribute in the entity with `@Id`.

The main purpose of the `IdClass` is to be used as the structure passed to the `EntityManager` `find()` and `getReference()` API. Some JPA products also use the `IdClass` as a cache key to track an object's identity. Because of this, it is required (depending on JPA product) to implement an `equals()` and `hashCode()` method on the `IdClass`. Ensure that the `equals()` method checks each part of the primary key, and correctly uses `equals` for objects and `==` for primitives. Ensure that the `hashCode()` method will return the same value for two equal objects.

TopLink<sup>2</sup> / EclipseLink<sup>3</sup> : Do not require the implementation of `equals()` or `hashCode()` in the id class.

#### Example id class annotation

```
...
@Entity
@IdClass(EmployeePK.class)
public class Employee {
    @Id
    private long employeeId

    @Id
    private long companyId

    @Id
    private long departmentId
    ...
}
```

---

1 <https://java.sun.com/javase/5/docs/api/javax/persistence/IdClass.html>

2 Chapter 11 on page 25

3 Chapter 10 on page 23

## Example id class XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <id-class class="org.acme.EmployeePK"/>
  <attributes>
    <id name="employeeId"/>
    <id name="companyId"/>
    <id name="departmentId"/>
  </attributes>
</entity>
```

## Example id class

```
...
public class EmployeePK {
    private long employeeId;

    private long companyId;

    private long departmentId;

    public EmployeePK(long employeeId, long companyId, long
departmentId) {
        this.employeeId = employeeId;
        this.companyId = companyId;
        this.departmentId = departmentId;
    }

    public boolean equals(Object object) {
        if (object instanceof EmployeePK) {
            EmployeePK pk = (EmployeePK)object;
            return employeeId == pk.employeeId && companyId ==
pk.companyId && departmentId == pk.departmentId;
        } else {
            return false;
        }
    }

    public int hashCode() {
        return employeeId + companyId + departmentId;
    }
}
```

### 23.1.2 Embedded Id

An `EmbeddedId` defines a separate `Embeddable` Java class to contain the entities primary key. It is defined through the `@EmbeddedId`<sup>4</sup> annotation or `<embedded-id>` XML element. The `EmbeddedId`'s `Embeddable` class must define each id attribute for the entity using `Basic` mappings. All attributes in the `EmbeddedId`'s `Embeddable` are assumed to be part of the primary key.

The `EmbeddedId` is also used as the structure passed to the `EntityManager` `find()` and `getReference()` API. Some JPA products also use the `EmbeddedId` as a cache key to track an object's identity. Because of this, it is required (depending on JPA product) to

<sup>4</sup> <https://java.sun.com/javaee/5/docs/api/javax/persistence/EmbeddedId.html>

implement an `equals()` and `hashCode()` method on the `EmbeddedId`. Ensure that the `equals()` method checks each part of the primary key, and correctly uses `equals` for objects and `==` for primitives. Ensure that the `hashCode()` method will return the same value for two equal objects.

TopLink<sup>5</sup> / EclipseLink<sup>6</sup> : Do not require the implementation of `equals()` or `hashCode()` in the id class.

### Example embedded id annotation

```
...
@Entity
public class Employee {
    @EmbeddedId
    private EmployeePK id
    ...
}
```

### Example embedded id XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
    <embedded-id class="org.acme.EmployeePK"/>
</entity>
<embeddable name="EmployeePK" class="org.acme.EmployeePK"
access="FIELD">
    <attributes>
        <basic name="employeeId"/>
        <basic name="companyId"/>
        <basic name="departmentId"/>
    </attributes>
</embeddable/>
```

### Example embedded id class

```
...
@Embeddable
public class EmployeePK {
    @Basic
    private long employeeId

    @Basic
    private long companyId

    @Basic
    private long departmentId

    public EmployeePK(long employeeId, long companyId, long
departmentId) {
        this.employeeId = employeeId;
        this.companyId = companyId;
        this.departmentId = departmentId;
    }
}
```

---

<sup>5</sup> Chapter 11 on page 25

<sup>6</sup> Chapter 10 on page 23

```
public boolean equals(Object object) {
    if (object instanceof EmployeePK) {
        EmployeePK pk = (EmployeePK)object;
        return employeeId == pk.employeeId && companyId ==
pk.companyId && departmentId == pk.departmentId;
    } else {
        return false;
    }
}

public int hashCode() {
    return employeeId + companyId + departmentId;
}
}
```

## 23.2 Primary Keys through OneToOne and ManyToOne Relationships

A common model is to have a dependent object share the primary key of its parent. In the case of a **OneToOne** the child's primary key is the same as the parent, and in the case of a **ManyToOne** the child's primary key is composed of the parent's primary key and another locally unique field.

JPA 1.0 does not allow `@Id` on a **OneToOne** or **ManyToOne**, but JPA 2.0 does.

One of the biggest pitfalls when working with composite primary keys comes with the implementation of associations to entity classes whose tables have multi-column primary keys (using the `@JoinColumns` annotation). Many JPA implementations may throw seemingly inconsistent exceptions when not specifying `referencedColumnName` for *every* `@JoinColumn` annotation, which JPA requires (even if all referenced column names are equal to the ones in the referencing table). See <http://download.oracle.com/javaee/5/api/javax/persistence/JoinColumns.html>

### 23.2.1 JPA 1.0

Unfortunately JPA 1.0 does not handle this model well, and things become complicated, so to make your life a little easier you may consider defining a generated unique id for the child. JPA 1.0 requires that all `@Id` mappings be **Basic** mappings, so if your `Id` comes from a foreign key column through a **OneToOne** or **ManyToOne** mapping, you must also define a **Basic** `@Id` mapping for the foreign key column. The reason for this is in part that the `Id` must be a simple object for identity and caching purposes, and for use in the `IdClass` or the `EntityManager find()` API.

Because you now have two mappings for the same foreign key column you must define which one will be written to the database (it must be the **Basic** one), so the **OneToOne** or **ManyToOne** foreign key must be defined to be read-only. This is done through setting the `JoinColumn` attributes `insertable` and `updatable` to false, or by using the `@PrimaryKeyJoinColumn` instead of the `@JoinColumn`.

A side effect of having two mappings for the same column is that you now have to keep the two in synch. This is typically done through having the set method for the **OneToOne**

attribute also set the **Basic** attribute value to the target object's id. This can become very complicated if the target object's primary key is a **GeneratedValue**, in this case you must ensure that the target object's id has been assigned *before* relating the two objects.

Some times I think that JPA primary keys would be much simpler if they were just defined on the entity using a collection of **Columns** instead of mixing them up with the attribute mapping. This would leave you free to map the primary key field in any manner you desired. A generic **List** could be used to pass the primary key to **find()** methods, and it would be the JPA provider's responsibility for hashing and comparing the primary key correctly instead of the user's **IdClass**. But perhaps for simple singleton primary key models the JPA model is more straight forward.

TopLink<sup>7</sup> / EclipseLink<sup>8</sup> : Allow the primary key to be specified as a list of columns instead of using Id mappings. This allows **OneToOne** and **ManyToOne** mapping foreign keys to be used as the primary key without requiring a duplicate mapping. It also allows the primary key to be defined through any other mapping type. This is set through using a **DescriptorCustomizer** and the **ClassDescriptor** **addPrimaryKeyFieldName** API.

Hibernate<sup>9</sup> / Open JPA<sup>10</sup> / EclipseLink<sup>11</sup> (as of 1.2): Allows the **@Id** annotation to be used on a **OneToOne** or **ManyToOne** mapping.

### Example OneToOne id annotation

```
...
@Entity
public class Address {
    @Id
    @Column(name="OWNER_ID")
    private long ownerId;

    @OneToOne
    @PrimaryKeyJoinColumn(name="OWNER_ID",
referencedColumnName="EMP_ID")
    private Employee owner;
    ...

    public void setOwner(Employee owner) {
        this.owner = owner;
        this.ownerId = owner.getId();
    }
    ...
}
```

### Example OneToOne id XML

```
<entity name="Address" class="org.acme.Address" access="FIELD">
  <attributes>
    <id name="ownerId">
```

---

7 Chapter 11 on page 25

8 Chapter 10 on page 23

9 Chapter 12 on page 27

10 Chapter 15 on page 33

11 Chapter 10 on page 23

```

        <column name="OWNER_ID"/>
    </id>
    <one-to-one name="owner">
        <primary-key-join-column name="OWNER_ID"
referencedColumnName="EMP_ID"/>
    </one-to-one>
</attributes>
<entity/>

```

### Example ManyToOne id annotation

```

...
@Entity
@IdClass({PhonePK.class})
public class Phone {
    @Id
    @Column(name="OWNER_ID")
    private long ownerId;

    @Id
    private String type;

    @ManyToOne
    @PrimaryKeyJoinColumn(name="OWNER_ID",
referencedColumnName="EMP_ID")
    private Employee owner;
    ...

    public void setOwner(Employee owner) {
        this.owner = owner;
        this.ownerId = owner.getId();
    }
    ...
}

```

### Example ManyToOne id XML

```

<entity name="Address" class="org.acme.Address" access="FIELD">
    <id-class class="org.acme.PhonePK"/>
    <attributes>
        <id name="ownerId">
            <column name="OWNER_ID"/>
        </id>
        <id name="type"/>
        <many-to-one name="owner">
            <primary-key-join-column name="OWNER_ID"
referencedColumnName="EMP_ID"/>
        </many-to-one>
    </attributes>
</entity>

```

## 23.2.2 JPA 2.0

Defining an Id for a `OneToOne` or `ManyToOne` in JPA 2.0 is much simpler. The `@Id` annotation or id XML attribute can be added to a `OneToOne` or `ManyToOne` mapping. The Id used for the object will be derived from the target object's Id. If the Id is a single value, then the source object's Id is the same as the target object's Id. If it is a composite Id, then the

`IdClass` will contain the `Basic` `Id` attributes, and the target object's `Id` as the relationship value. If the target object also has a composite `Id`, then the source object's `IdClass` will contain the target object's `IdClass`.

### Example JPA 2.0 `ManyToOne` `id` annotation

```
...
@Entity
@IdClass({PhonePK.class})
public class Phone {

    @Id
    private String type;

    @ManyToOne
    @Id
    @JoinColumn(name="OWNER_ID", referencedColumnName="EMP_ID")
    private Employee owner;
    ...//getters and setters
}
```

### Example JPA 2.0 `ManyToOne` `id` XML

```
<entity name="Address" class="org.acme.Address" access="FIELD">
  <id-class class="org.acme.PhonePK"/>
  <attributes>
    <id name="type"/>
    <many-to-one name="owner" id="true">
      <join-column name="OWNER_ID"
referencedColumnName="EMP_ID"/>
    </many-to-one>
  </attributes>
</entity>
```

### Example JPA 2.0 `id` class

```
...
public class PhonePK {
    private String type;
    private long owner;

    public PhonePK() {}

    public PhonePK(String type, long owner) {
        this.type = type;
        this.owner = owner;
    }

    public boolean equals(Object object) {
        if (object instanceof PhonePK) {
            PhonePK pk = (PhonePK)object;
            return type.equals(pk.type) && owner == pk.owner;
        } else {
            return false;
        }
    }
}
```

```

    public int hashCode() {
        return type.hashCode() + owner;
    }
}

```

## 23.3 Advanced Sequencing

### Concurrency and Deadlocks

One issue with table sequencing is that the sequence table can become a concurrency bottleneck, even causing deadlocks. If the sequence ids are allocated in the same transaction as the insert, this can cause poor concurrency, as the sequence row will be locked for the duration of the transaction, preventing any other transaction that needs to allocate a sequence id. In some cases the entire sequence table or the table page could be locked causing even transactions allocating other sequences to wait or even deadlock. If a large sequence pre-allocation size is used this becomes less of an issue, because the sequence table is rarely accessed. Some JPA providers use a separate (non-JTA) connection to allocate the sequence ids in, avoiding or limiting this issue. In this case, if you use a JTA data-source connection, it is important to also include a non-JTA data-source connection in your persistence.xml.

### Guaranteeing Sequential Ids

Table sequencing also allows for truly sequential ids to be allocated. Sequence and identity sequencing are non-transactional and typically cache values on the database, leading to large gaps in the ids that are allocated. Typically this is not an issue and desired to have good performance, however if performance and concurrency are less of a concern, and true sequential ids are desired then a table sequence can be used. By setting the `allocationSize` of the sequence to 1 and ensuring the sequence ids are allocated in the same transaction of the insert, you can guarantee sequence ids without gaps (but generally it is much better to live with the gaps and have good performance).

#### 23.3.1 Running Out of Numbers

One paranoid delusional fear that programmers frequently have is running out of sequence numbers. Since most sequence strategies just keep incrementing a number it is unavoidable that you will eventually run out. However as long as a large enough numeric precision is used to store the sequence id this is not an issue. For example if you stored your id in a `NUMBER(5)` column, this would allow 99,999 different ids, which on most systems would eventually run out. However if you store your id in a `NUMBER(10)` column, which is more typical, this would store 9,999,999,999 ids, or one id each second for about 300 years (longer than most databases exist). But perhaps your system will process a lot of data, and (hopefully) be around a very long time. If you store your id in a `NUMBER(20)` this would be 99,999,999,999,999,999,999 ids, or one id each millisecond for about 3,000,000,000 years, which is pretty safe.



But you also need to store this id in Java. If you store the id in a Java int, this would be a 32 bit number, which is 4,294,967,296 different ids (well actually 2,147,483,648 positive ids), or one id each second for about 100 years. If you instead use a long, this would be a 64 bit number, which is 9,223,372,036,854,775,808 different ids, or one id each millisecond for about 300,000,000 years, which is pretty safe. I would recommend using a long vs an int however, as I have seen int ids run out on large databases before (what happens is they become negative until they wrap around to 0 and start getting constraint errors).

### 23.3.2 Customizing

JPA supports three different strategies for generating ids, however there are many other methods. Normally the JPA strategies are sufficient, so you would only use a different method in a legacy situation.

Sometimes the application has an application specific strategy for generating ids, such as prefixing ids with the country code, or branch number. There are several ways to integrate a customize ids generation strategy, the simplest is just define the id as a normal id and have the application assign the id value when the object is created.

Some JPA products provide additional sequencing and id generation options, and configuration hooks.

TopLink<sup>12</sup>, EclipseLink<sup>13</sup> : Several additional sequencing options are provided. A `UnaryTableSequence` allows a single column table to be used. A `QuerySequence` allows for custom SQL or stored procedures to be used. An API also exists to allow a user to supply their own code for allocating ids.

Hibernate<sup>14</sup> : A GUID id generation options is provided through the `@GenericGenerator` annotation.

## 23.4 Primary Keys through Triggers

A database table can be defined to have a trigger that automatically assign its' primary key. Generally this is normally not a good idea (although some DBAs may think it is), and it is better to use a JPA provider generated sequence id, or assign the id in the application. The main issue with the id being assigned in a trigger is that the application and object require this value back. For non-primary key values assigned through triggers it is possible to refresh the object after committing or flushing the object to obtain the values back. However this is not possible for the id, as the id is required to refresh an object.

If you have an alternative way to select the id generated by the trigger, such as selecting the object's row using another unique field, you could issue this SQL select after the insert to obtain the id and set it back in the object. You could perform this select in a JPA `@PostPersist`<sup>15</sup> event. Some JPA providers may not allow/like a query execution during

---

<sup>12</sup> Chapter 11 on page 25

<sup>13</sup> Chapter 10 on page 23

<sup>14</sup> Chapter 12 on page 27

<sup>15</sup> <https://java.sun.com/javaee/5/docs/api/javax/persistence/PostPersist.html>

an event, they also may not pick up a change to an object during an event callback, so there may be issues with doing this. Also some JPA providers may not allow the primary key to be un-assigned/null when not using a `GeneratedValue`, so you may have issues. Some JPA providers have built-in support for returning values assigned in a trigger (or stored procedure) back into the object.

TopLink<sup>16</sup> / EclipseLink<sup>17</sup> : Provide a `ReturningPolicy` that allows for any field values including the primary key to be returned from the database after an insert or update. This is defined through the `@ReturnInsert`, `@ReturnUpdate` annotations, or the `<return-insert>`, `<return-update>` XML elements in the `eclipselink-orm.xml`.

## 23.5 Primary Keys through Events

If the application generates its' own id instead of using a JPA `GeneratedValue`, it is sometimes desirable to perform this id generation in a JPA event, instead of the application code having to generate and set the id. In JPA this can be done through the `@PrePersist`<sup>18</sup> event.

## 23.6 No Primary Key

Sometimes your object or table has no primary key. The best solution in this case is normally to add a generated id to the object and table. If you do not have this option, sometimes there is a column or set of columns in the table that make up a unique value. You can use this unique set of columns as your id in JPA. The JPA `Id` does not always have to match the database table primary key constraint, nor is a primary key or a unique constraint required.

If your table truly has no unique columns, then use all of the columns as the id. Typically when this occurs the data is read-only, so even if the table allows duplicate rows with the same values, the objects will be the same anyway, so it does not matter that JPA thinks they are the same object. The issue with allowing updates and deletes is that there is no way to uniquely identify the object's row, so all of the matching rows will be updated or deleted.

If your object does not have an id, but its' table does, this is fine. Make the object an `Embeddable` object, embeddable objects do not have ids. You will need a `Entity` that contains this `Embeddable` to persist and query it.

Identity and Sequencing<sup>19</sup> Identity and Sequencing<sup>20</sup>

---

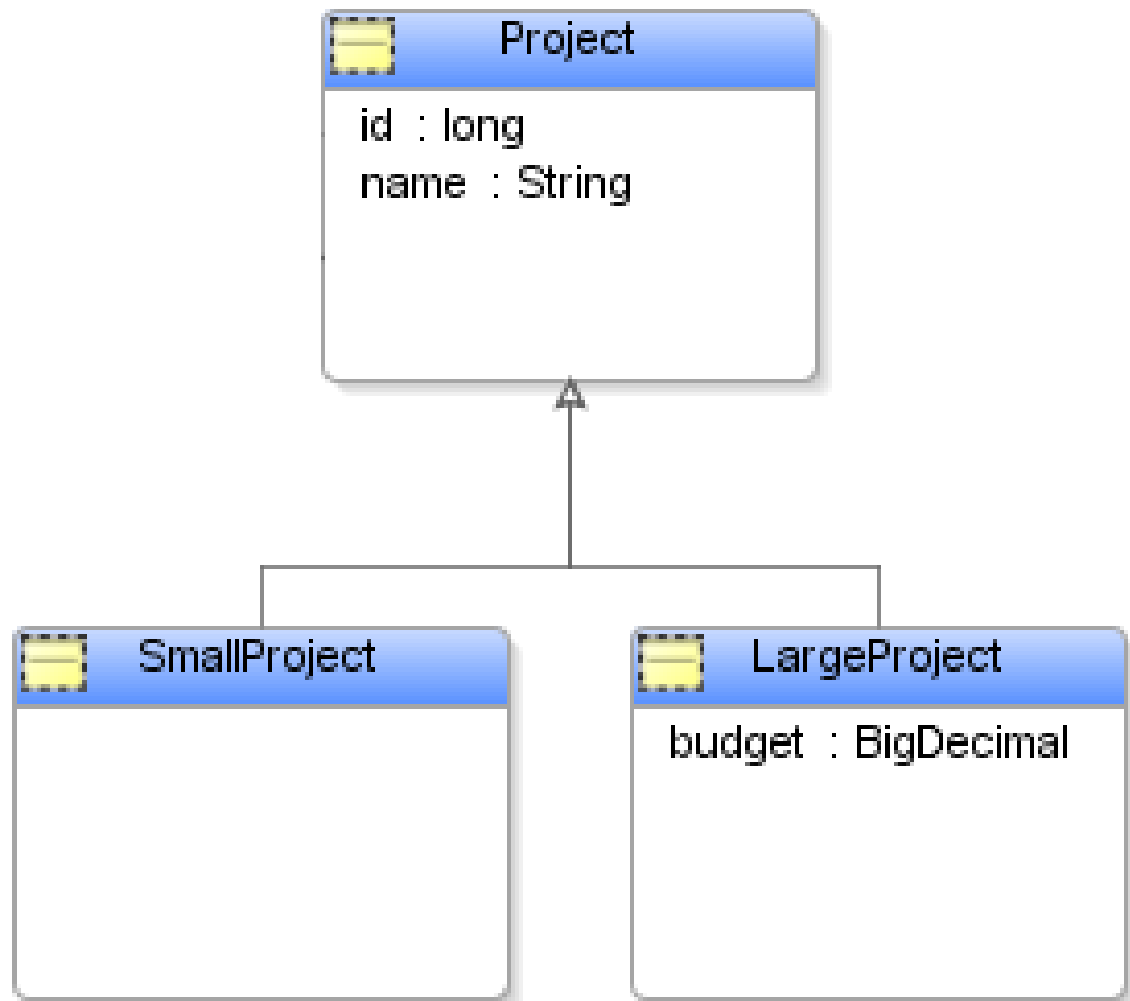
16 Chapter 11 on page 25

17 Chapter 10 on page 23

18 <https://java.sun.com/javase/5/docs/api/javax/persistence/PrePersist.html>

19 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

20 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FMapping>



**Figure 12** An example of inheritance. `SmallProject` and `LargeProject` inherit the properties of their common parent, `Project`.

Inheritance is a fundamental concept of object-oriented programming and Java. Relational databases have no concept of inheritance, so persisting inheritance in a database can be tricky. Because relational databases have no concept of inheritance, there is no standard way of implementing inheritance in database, so the hardest part of persisting inheritance is choosing how to represent the inheritance in the database.

JPA defines several inheritance mechanisms, mainly defined through the `@Inheritance`<sup>21</sup> annotation or the `<inheritance>` element. There are three inheritance strategies defined from the `InheritanceType`<sup>22</sup> enum, `SINGLE_TABLE`, `TABLE_PER_CLASS` and `JOINED`.

*Single table* inheritance is the default, and *table per class* is an *optional* feature of the JPA spec, so not all providers may support it. JPA also defines a mapped superclass concept defined

<sup>21</sup> <https://java.sun.com/javaee/5/docs/api/javax/persistence/Inheritance.html>

<sup>22</sup> <https://java.sun.com/javaee/5/docs/api/javax/persistence/InheritanceType.html>

though the `@MappedSuperclass`<sup>23</sup> annotation or the `<mapped-superclass>` element. A mapped superclass is not a persistent class, but allow common mappings to be define for its subclasses.

## 23.7 Single Table Inheritance

Single table inheritance is the simplest and typically the best performing and best solution. In single table inheritance a single table is used to store all of the instances of the entire inheritance hierarchy. The table will have a column for *every* attribute of *every* class in the hierarchy. A discriminator column is used to determine which class the particular row belongs to, each class in the hierarchy defines its own unique discriminator value.

### 23.7.1 Example single table inheritance table

PROJECT (table)

ID	PROJ_TYPE	NAME	BUDGET
1	L	Accounting	50000
2	S	Legal	null

### 23.7.2 Example single table inheritance annotations

```
@Entity
@Inheritance
@DiscriminatorColumn(name="PROJ_TYPE")
@Table(name="PROJECT")
public abstract class Project {
    @Id
    private long id;
    ...
}

@Entity
@DiscriminatorValue("L")
public class LargeProject extends Project {
    private BigDecimal budget;
}

@Entity
@DiscriminatorValue("S")
public class SmallProject extends Project {
}
```

### 23.7.3 Example single table inheritance XML

```
<entity name="Project" class="org.acme.Project" access="FIELD">
```

23 <https://java.sun.com/javase/5/docs/api/javax/persistence/MappedSuperclass.html>

```
<table name="PROJECT"/>
<inheritance/>
<discriminator-column name="PROJ_TYPE"/>
<attributes>
  <id name="id"/>
  ...
</attributes>
<entity/>

<entity name="LargeProject" class="org.acme.LargeProject"
access="FIELD">
  <discriminator-value>L</discriminator-value>
  ...
<entity/>

<entity name="SmallProject" class="org.acme.SmallProject"
access="FIELD">
  <discriminator-value>S</discriminator-value>
<entity/>
```

### 23.7.4 Common Problems

#### *No class discriminator column*

If you are mapping to an existing database schema, your table may not have a class discriminator column. Some JPA providers do not require a class discriminator when using a *joined* inheritance strategy, so this may be one solution. Otherwise you need some way to determine the class for a row. Sometimes the inherited value can be computed from several columns, or there is an discriminator but not a one to one mapping from value to class. Some JPA providers provide extended support for this. Another option is to create a database view that manufactures the discriminator column, and then map your hierarchy to this view instead of the table. In general the best solution is just to add a discriminator column to the table (truth be told, ALTER TABLE is your best friend in ORM).

TopLink<sup>24</sup> / EclipseLink<sup>25</sup> : Support computing the inheritance discriminator through Java code. This can be done through using a `DescriptorCustomizer` and the `ClassDescriptor`'s `InheritancePolicy`'s `setClassExtractor()` method.

Hibernate<sup>26</sup> : This can be accomplished through using the Hibernate `@DiscriminatorFormula` annotation. This allows database specific SQL or functions to be used to compute the discriminator value.

#### *Non nullable attributes*

Subclasses cannot define attributes as not allowing null, as the other subclasses must insert null into those columns. A workaround to this issue is instead of defining a not null constraint on the column, define a table constraint that check the discriminator value and

---

<sup>24</sup> Chapter 11 on page 25

<sup>25</sup> Chapter 10 on page 23

<sup>26</sup> Chapter 12 on page 27

the not nullable value. In general the best solution is to just live without the constraint (odds are you have enough constraints in your life to deal with as it is).

## 23.8 Joined, Multiple Table Inheritance

Joined inheritance is the most logical inheritance solution because it mirrors the object model in the data model. In joined inheritance a table is defined for *each* class in the inheritance hierarchy to store *only* the local attributes of that class. Each table in the hierarchy must also store the object's id (primary key), which is *only* defined in the root class. All classes in the hierarchy must share the same id attribute. A discriminator column is used to determine which class the particular row belongs to, each class in the hierarchy defines its own unique discriminator value.

Some JPA providers support joined inheritance with or without a discriminator column, some required the discriminator column, and some do not support the discriminator column. So joined inheritance does not seem to be fully standardized yet.

Hibernate<sup>27</sup> : A discriminator column on joined inheritance is not supported. <http://opensource.atlassian.com/projects/hibernate/browse/ANN-140>

### 23.8.1 Example joined inheritance tables

PROJECT (table)

ID	PROJ__TYPE	NAME
1	L	Accounting
2	S	Legal

SMALLPROJECT (table)

ID
2

LARGEPROJECT (table)

ID	BUDGET
1	50000

### 23.8.2 Example joined inheritance annotations

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="PROJ__TYPE")
```

```
@Table(name="PROJECT")
public abstract class Project {
    @Id
    private long id;
    ...
}

@Entity
@DiscriminatorValue("L")
@Table(name="LARGEPROJECT")
public class LargeProject extends Project {
    private BigDecimal budget;
}

@Entity
@DiscriminatorValue("S")
@Table(name="SMALLPROJECT")
public class SmallProject extends Project {
}
```

### 23.8.3 Example joined inheritance XML

```
<entity name="Project" class="org.acme.Project" access="FIELD">
    <table name="PROJECT"/>
    <inheritance strategy="JOINED"/>
    <discriminator-column name="PROJ_TYPE"/>
    <attributes>
        <id name="id"/>
        ...
    </attributes>
</entity>

<entity name="LargeProject" class="org.acme.LargeProject"
access="FIELD">
    <table name="LARGEPROJECT"/>
    <discriminator-value>L</discriminator-value>
    ...
</entity>

<entity name="SmallProject" class="org.acme.SmallProject"
access="FIELD">
    <table name="SMALLPROJECT"/>
    <discriminator-value>S</discriminator-value>
</entity>
```

### 23.8.4 Common Problems

#### *Poor query performance*

The main disadvantage to the joined model is that to query any class join queries are required. Querying the root or branch classes is even more difficult as either multiple queries are required, or outer joins or unions are required. One solution is to use single table inheritance instead, this is good if the classes have a lot in common, but if it is a big hierarchy and the subclasses have little in common this may not be desirable. Another solution is to remove the inheritance and instead use a `MappedSuperclass`, but this means that you can no longer query or have relationships to the class.

The poorest performing queries will be those to the root or branch classes. Avoiding queries and relationships to the root and branch classes will help to alleviate this burden. If you must query the root or branch classes there are two methods that JPA providers use, one is to outer join all of the subclass tables, the second is to first query the root table, then query only the required subclass table directly. The first method has the advantage of only requiring one query, the second has the advantage of avoiding outer joins which typically have poor performance in databases. You may wish to experiment with each to determine which mechanism is more efficient in your application and see if your JPA provider supports that mechanism. Typically the multiple query mechanism is more efficient, but this generally depends on the speed of your database connection.

TopLink<sup>28</sup> / EclipseLink<sup>29</sup> : Support both querying mechanisms. The multiple query mechanism is used by default. Outer joins can be used instead through using a `DescriptorCustomizer` and the `ClassDescriptor`'s `InheritancePolicy`'s `setShouldOuterJoinSubclasses()` method.

### ***Do not have/want a table for every subclass***

Most inheritance hierarchies do not fit with either the *joined* or the *single table* inheritance strategy. Typically the desired strategy is somewhere in between, having joined tables in some subclasses and not in others. Unfortunately JPA does not directly support this. One workaround is to map your inheritance hierarchy as single table, but then add the additional tables in the subclasses, either through defining a `Table` or `SecondaryTable` in each subclass as required. Depending on your JPA provider, this may work (don't forget to sacrifice the chicken). If it does not work, then you may need to use a JPA provider specific solution if one exists for your provider, otherwise live within the constraints of having either a single table or one per subclass. You could also change your inheritance hierarchy so it matches your data model, so if the subclass does not have a table, then collapse its' class into its' superclass.

### ***No class discriminator column***

If you are mapping to an existing database schema, your table may not have a class discriminator column. Some JPA providers do not require a class discriminator when using a *joined* inheritance strategy, so this may be one solution. Otherwise you need some way to determine the class for a row. Sometimes the inherited value can be computed from several columns, or there is an discriminator but not a one to one mapping from value to class. Some JPA providers provide extended support for this. Another option is to create a database view that manufactures the discriminator column, and then map your hierarchy to this view instead of the table.

---

28 Chapter 11 on page 25

29 Chapter 10 on page 23



TopLink<sup>30</sup> / EclipseLink<sup>31</sup> : Support computing the inheritance discriminator through Java code. This can be done through using a `DescriptorCustomizer` and the `ClassDescriptor`'s `InheritancePolicy`'s `setClassExtractor()` method.

Hibernate<sup>32</sup> : This can be accomplished through using the Hibernate `@DiscriminatorFormula` annotation. This allows database specific SQL or functions to be used to compute the discriminator value.

---

<sup>30</sup> Chapter 11 on page 25

<sup>31</sup> Chapter 10 on page 23

<sup>32</sup> Chapter 12 on page 27

## 24 Advanced

### 24.1 Table Per Class Inheritance

Table per class inheritance allows inheritance to be used in the object model, when it does not exist in the data model. In table per class inheritance a table is defined for *each* concrete class in the inheritance hierarchy to store *all* the attributes of that class and *all* of its superclasses. Be cautious using this strategy as it is optional in the JPA spec, and querying root or branch classes can be very difficult and inefficient.

#### 24.1.1 Example table per class inheritance tables

SMALLPROJECT (table)

ID	NAME
2	Legal

LARGEPROJECT (table)

ID	NAME	BUDGET
1	Accounting	50000

#### 24.1.2 Example table per class inheritance annotations

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Project {
    @Id
    private long id;
    ...
}
```

```
@Entity
@Table(name="LARGEPROJECT")
public class LargeProject extends Project {
    private BigDecimal budget;
}
```

```
@Entity
@Table(name="SMALLPROJECT")
public class SmallProject extends Project {
}
```

### 24.1.3 Example table per class inheritance XML

```
<entity name="Project" class="org.acme.Project" access="FIELD">
  <inheritance strategy="TABLE_PER_CLASS"/>
  <attributes>
    <id name="id"/>
    ...
  </attributes>
</entity>

<entity name="LargeProject" class="org.acme.LargeProject"
access="FIELD">
  <table name="LARGEPROJECT"/>
  ...
</entity>

<entity name="SmallProject" class="org.acme.SmallProject"
access="FIELD">
  <table name="SMALLPROJECT"/>
</entity>
```

### 24.1.4 Common Problems

#### *Poor query performance*

The main disadvantage to the table per class model is queries or relationships to the root or branch classes become expensive. Querying the root or branch classes require multiple queries, or unions. One solution is to use single table inheritance instead, this is good if the classes have a lot in common, but if it is a big hierarchy and the subclasses have little in common this may not be desirable. Another solution is to remove the table per class inheritance and instead use a **MappedSuperclass**, but this means that you can no longer query or have relationships to the class.

#### *Issues with ordering and joins*

Because table per class inheritance requires multiple queries, or unions, you cannot join to, fetch join, or traverse them in queries. Also when ordering is used the results will be ordered by class, then by the ordering. These limitations depend on your JPA provider, some JPA provider may have other limitations, or not support table per class at all as it is optional in the JPA spec.

## 24.2 Mapped Superclasses

Mapped superclass inheritance allows inheritance to be used in the object model, when it does not exist in the data model. It is similar to table per class inheritance, but does not allow querying, persisting, or relationships to the superclass. Its' main purpose is to allow mappings information to be inherited by its' subclasses. The subclasses are responsible for defining the table, id and other information, and can modify any of the inherited mappings. A common usage of a mapped superclass is to define a common **PersistentObject** for your

application to define common behavior and mappings such as the id and version. A mapped superclass normally should be an abstract class. A mapped superclass is *not* an `Entity` but is instead defined through the `@MappedSuperclass`<sup>1</sup> annotation or the `<mapped-superclass>` element.

### 24.2.1 Example mapped superclass tables

SMALLPROJECT (table)

ID	NAME
2	Legal

LARGEPROJECT (table)

ID	PROJECT_NAME	BUDGET
1	Accounting	50000

### 24.2.2 Example mapped superclass annotations

```
@MappedSuperclass
public abstract class Project {
    @Id
    private long id;
    @Column(name="NAME")
    private String name;
    ...
}

@Entity
@Table(name="LARGEPROJECT")
@AttributeOverride(name="name", column=@Column(name="PROJECT_NAME"))
public class LargeProject extends Project {
    private BigDecimal budget;
}

@Entity
@Table("SMALLPROJECT")
public class SmallProject extends Project {
}
```

### 24.2.3 Example mapped superclass XML

```
<mapped-superclass class="org.acme.Project" access="FIELD">
  <attributes>
    <id name="id"/>
    <basic name="name">
      <column name="NAME"/>
    </basic>
  </attributes>
</mapped-superclass>
```

1 <https://java.sun.com/javase/5/docs/api/javax/persistence/MappedSuperclass.html>

```
    ...
  </attributes>
</mapped-superclass/>

<entity name="LargeProject" class="org.acme.LargeProject"
  access="FIELD">
  <table name="LARGEPROJECT"/>
  <attribute-override>
    <column name="NAME"/>
  </attribute-override>
  ...
</entity/>

<entity name="SmallProject" class="org.acme.SmallProject"
  access="FIELD">
  <table name="SMALLPROJECT"/>
</entity/>
```

## 24.2.4 Common Problems

### *Cannot query, persist, or have relationships*

The main disadvantage of mapped superclasses is that they cannot be queried or persisted. You also cannot have a relationship to a mapped superclass. If you require any of these then you must use another inheritance model, such as table per class, which is virtually identical to a mapped superclass except it (may) not have these limitations. Another alternative is to change your model such that your classes do not have relationships to the superclass, such as changing the relationship to a subclass, or removing the relationship and instead querying for its value by querying each possible subclass and collecting the results in Java.

### *Subclass does not want to inherit mappings*

Sometimes you have a subclass that needs to be mapped differently than its parent, or is similar to its' parent but does not have one of the fields, or uses it very differently. Unfortunately it is very difficult not to inherit everything from your parent in JPA, you can override a mapping, but you cannot remove one, or change the type of mapping, or the target class. If you define your mappings as properties (get methods), or through XML, you may be able to attempt to override or mark the inherited mapping as **Transient**, this may work depending on your JPA provider (don't forget to sacrifice a chicken).

Another solution is to actually fix your inheritance in your object model. If you inherit `foo` from `Bar` but don't want to inherit it, then remove it from `Bar`, if the other subclasses need it, either add it to each, or create a `FooBar` subclass of `Bar` that has the `foo` and have the other subclasses extend this.

Some JPA providers may provide ways to be less stringent on inheritance.

TopLink<sup>2</sup> / EclipseLink<sup>3</sup> : Allow a subclass remove a mapping, redefine a mapping, or be entirely independent of its superclass. This can be done through using a `DescriptorCustomizer` and removing the `ClassDescriptor`'s mapping, or adding a mapping with the same attribute name, or removing the `InheritancePolicy`.

Inheritance<sup>4</sup> Inheritance<sup>5</sup>

---

<sup>2</sup> Chapter 11 on page 25

<sup>3</sup> Chapter 10 on page 23

<sup>4</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

<sup>5</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FMapping>



## 25 Embeddables

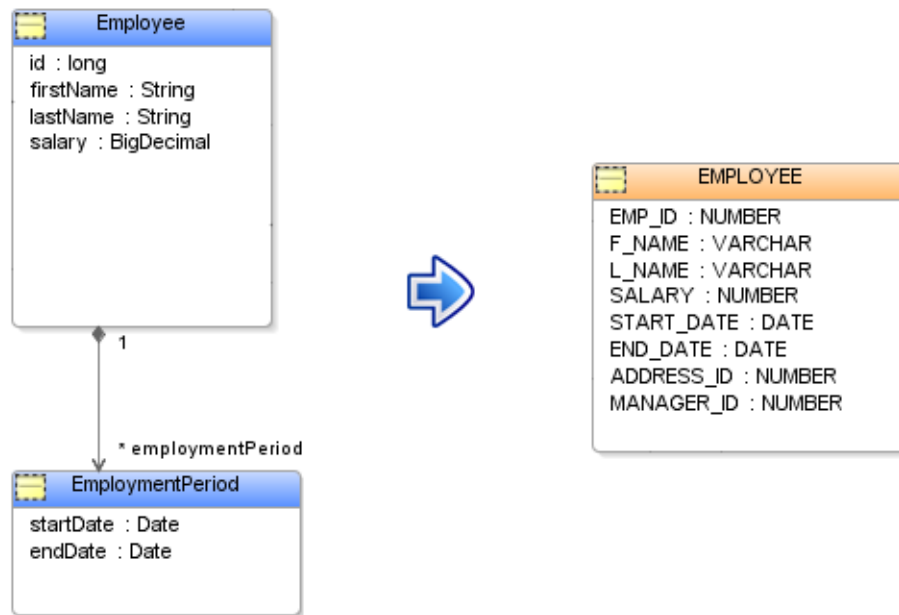


Figure 13

In an application object model some objects are considered independent, and others are considered dependent parts of other objects. In UML a relationship to a dependent object is considered an aggregate or composite association. In a relational database this kind of relationship could be modeled in two ways, the dependent object could have its own table, or its data could be *embedded* in the independent object's table.

In JPA a relationship where the target object's data is embedded in the source object's table is considered an embedded relationship, and the target object is considered an Embeddable object. Embeddable objects have different requirements and restrictions than Entity objects and are defined by the `@Embeddable`<sup>1</sup> annotation or `<embeddable>` element.

An embeddable object cannot be directly persisted, or queried<sup>2</sup>, it can only be persisted or queried in the context of its parent. An embeddable object does not have an id or table.

<sup>1</sup> <https://java.sun.com/javase/5/docs/api/javax/persistence/Embeddable.html>

<sup>2</sup> Chapter 26.8 on page 98



The JPA spec does not support embeddable objects having inheritance<sup>3</sup>, although some JPA providers may allow this. Relationships<sup>4</sup> from embeddable objects are supported in JPA 2.0.

Relationships to embeddable objects are defined through the `@Embedded`<sup>5</sup> annotation or `<embedded>` element. The JPA 2.0 spec also supports collection<sup>6</sup> relationships to embeddable objects.

### 25.0.5 Example of an Embeddable object annotations

```
@Embeddable
public class EmploymentPeriod {
    @Column(name="START_DATE")
    private java.sql.Date startDate;

    @Column(name="END_DATE")
    private java.sql.Date endDate;
    ...
}
```

### 25.0.6 Example of an Embeddable object XML

```
<embeddable class="org.acme.EmploymentPeriod" access="FIELD">
  <attributes>
    <basic name="startDate">
      <column name="START_DATE"/>
    </basic>
    <basic name="endDate">
      <column name="END_DATE"/>
    </basic>
  </attributes>
</embeddable>
```

### 25.0.7 Example of an embedded relationship annotations

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @Embedded
    private EmploymentPeriod period;
    ...
}
```

### 25.0.8 Example of an embedded relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
```

---

3 Chapter 26.5 on page 96

4 Chapter 26.6 on page 96

5 <https://java.sun.com/javase/5/docs/api/javax/persistence/Embedded.html>

6 Chapter 26.7 on page 97

```
<attributes>
  <id name="id"/>
  <embedded name="period"/>
</attributes>
</entity>
```



## 26 Advanced

### 26.1 Sharing

An embeddable object can be shared between multiple classes. Consider a `Name` object, that both an `Employee` and a `User` contain. Both `Employee` and a `User` have their own tables, with different column names that they desire to store their name in. Embeddables support this through allowing each embedded mapping to override the columns used in the embeddable. This is done through the `@AttributeOverride`<sup>1</sup> annotation or `<attribute-override>` element.

Note that an embeddable cannot be shared between multiple instances. If you desire to share an embeddable object instance, then you must make it an independent object with its own table.

#### 26.1.1 Example shared embeddable annotations

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="startDate",
            column=@Column(name="START_DATE")),
        @AttributeOverride(name="endDate",
            column=@Column(name="END_DATE"))
    })
    private Period employmentPeriod;
    ...
}

@Entity
public class User {
    @Id
    private long id;
    ...
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="startDate",
            column=@Column(name="SDATE")),
        @AttributeOverride(name="endDate", column=@Column(name="EDATE"))
    })
    private Period period;
    ...
}
```

---

<sup>1</sup> <https://java.sun.com/javase/5/docs/api/javax/persistence/AttributeOverride.html>

### 26.1.2 Example shared embeddable XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <embedded name="employmentPeriod">
      <attribute-override name="startDate">
        <column name="START_DATE"/>
      </attribute-override>
      <attribute-override name="endDate">
        <column name="END_DATE"/>
      </attribute-override>
    </embedded>
  </attributes>
</entity>
<entity name="User" class="org.acme.User" access="FIELD">
  <attributes>
    <id name="id"/>
    <embedded name="period">
      <attribute-override name="startDate">
        <column name="SDATE"/>
      </attribute-override>
      <attribute-override name="endDate">
        <column name="EDATE"/>
      </attribute-override>
    </embedded>
  </attributes>
</entity>
```

## 26.2 Embedded Ids

An `EmbeddedId` is an embeddable object that contains the `Id` for an entity.

See: [Embedded Id](#)<sup>2</sup>

## 26.3 Nulls

An embeddable object's data is contained in several columns in its parent's table. Since there is no single field value, there is no way to know if a parent's reference to the embeddable is `null`. One could assume that if every field value of the embeddable is `null`, then the reference should be `null`, but then there is no way to represent an embeddable with all `null` values. JPA does not allow embeddables to be `null`, but some JPA providers may support this.

[TopLink](#)<sup>3</sup> / [EclipseLink](#)<sup>4</sup> : Support an embedded reference being `null`. This is set through using a `DescriptorCustomizer` and the `AggregateObjectMapping` `setIsNullAllowed` API.

---

<sup>2</sup> Chapter 23.1.2 on page 67

<sup>3</sup> Chapter 11 on page 25

<sup>4</sup> Chapter 10 on page 23

## 26.4 Nesting

A nested embeddable is a relationship to an embeddable object from another embeddable. The JPA 1.0 spec only allows **Basic** relationships in an embeddable object, so nested embeddables are not supported, however some JPA products may support them. Technically there is nothing preventing the `@Embedded` annotation being used in an embeddable object, so this may just work depending on your JPA provider (cross your fingers).

JPA 2.0 supports nested embeddable objects.

TopLink<sup>5</sup> / EclipseLink<sup>6</sup> : Support embedded mappings from embeddables. The existing `@Embedded` annotation or `<embedded>` element can be used.

A workaround to having a nested embeddable, and for embeddables in general is to use property access, and add get/set methods for all of the attributes of the nested embeddable object.

### Example of using properties to define a nested embeddable

```
@Embeddable
public class EmploymentDetails {
    private EmploymentPeriod period;
    private int yearsOfService;
    private boolean fullTime;
    ....
    public EmploymentDetails() {
        this.period = new EmploymentPeriod();
    }
    @Transient
    public EmploymentPeriod getEmploymentPeriod() {
        return period;
    }
    @Basic
    public Date getStartDate() {
        return getEmploymentPeriod().getStartDate();
    }
    public void setStartDate(Date startDate) {
        getEmploymentPeriod().setStartDate(startDate);
    }
    @Basic
    public Date getEndDate() {
        return getEmploymentPeriod().getEndDate();
    }
    public void setEndDate(Date endDate) {
        getEmploymentPeriod().setEndDate(endDate);
    }
    ....
}
```

---

<sup>5</sup> Chapter 11 on page 25

<sup>6</sup> Chapter 10 on page 23

## 26.5 Inheritance

Embeddable inheritance is when one embeddable class subclasses another embeddable class. The JPA spec does not allow inheritance in embeddable objects, however some JPA products may support this. Technically there is nothing preventing the `@DiscriminatorColumn` annotation being used in an embeddable object, so this may just work depending on your JPA provider (cross your fingers). Inheritance in embeddables is always *single table* as an embeddable must live within its' parent's table. Generally attempting to mix inheritance between embeddables and entities is not a good idea, but may work in some cases.

TopLink<sup>7</sup> / EclipseLink<sup>8</sup> : Support inheritance with embeddables. This is set through using a `DescriptorCustomizer` and the `InheritancePolicy`.

## 26.6 Relationships

A relationship is when an embeddable has a `OneToOne` or other such mapping to an entity. The JPA 1.0 spec only allows **Basic** mappings in an embeddable object, so relationships from embeddables are not supported, however some JPA products may support them. Technically there is nothing preventing the `@OneToOne` annotation or other relationships from being used in an embeddable object, so this may just work depending on your JPA provider (cross your fingers).

JPA 2.0 supports all relationship types from an embeddable object.

TopLink<sup>9</sup> / EclipseLink<sup>10</sup> : Support relationship mappings from embeddables. The existing relationship annotations or XML elements can be used.

Relationships to embeddable objects from entities other than the embeddable's parent are typically not a good idea, as an embeddable is a private dependent part of its parent. Generally relationships should be to the embeddable's parent, not the embeddable. Otherwise, it would normally be a good idea to make the embeddable an independent entity with its own table. If an embeddable has a bi-directional relationship, such as a `OneToMany` that requires an inverse `ManyToOne` the inverse relationship should be to the embeddable's parent.

A workaround to having a relationship from an embeddable is to define the relationship in the embeddable's parent, and define property get/set methods for the relationship that set the relationship into the embeddable.

### Example of setting a relationship in an embeddable from its parent

```
@Entity
public class Employee {
    ....
    private EmploymentDetails details;
```

---

7 Chapter 11 on page 25

8 Chapter 10 on page 23

9 Chapter 11 on page 25

10 Chapter 10 on page 23

```

....
@Embedded
public EmploymentDetails getEmploymentDetails() {
    return details;
}
@OneToOne
public Address getEmploymentAddress() {
    return getEmploymentDetails().getAddress();
}
public void setEmploymentAddress(Address address) {
    getEmploymentDetails().setAddress(address);
}
}

```

One special relationship that is sometimes desired in an embeddable is a relationship to its parent. JPA does not support this, but some JPA providers may.

Hibernate<sup>11</sup> : Supports a `@Parent` annotation in embeddables to define a relationship to its parent.

A workaround to having a parent relationship from an embeddable is to set the parent in the property set method.

### Example of setting a relationship in an embeddable to its parent

```

@Entity
public class Employee {
    ....
    private EmploymentDetails details;
    ....
    @Embedded
    public EmploymentDetails getEmploymentDetails() {
        return details;
    }
    public void setEmploymentDetails(EmploymentDetails details) {
        this.details = details;
        details.setParent(this);
    }
}

```

## 26.7 Collections

A collection of embeddable objects is similar to a `OneToMany` except the target objects are embeddables and have no `Id`. This allows for a `OneToMany` to be defined without a inverse `ManyToOne`, as the parent is responsible for storing the foreign key in the target object's table. JPA 1.0 did not support collections of embeddable objects, but some JPA providers support this.

JPA 2.0 does support collections of embeddable objects through the `ElementCollection` mapping. See, `ElementCollection`<sup>12</sup>.

EclipseLink<sup>13</sup>(as of 1.2) : Supports the JPA 2.0 `ElementCollection` mapping.

<sup>11</sup> Chapter 12 on page 27

<sup>12</sup> Chapter 39.1.2 on page 185

<sup>13</sup> Chapter 10 on page 23



TopLink<sup>14</sup> / EclipseLink<sup>15</sup> : Support collections of embeddables. This is set through using a `DescriptorCustomizer` and the `AggregateCollectionMapping`.

Hibernate<sup>16</sup> : Supports collections of embeddables through the `@CollectionOfElements` annotation.

DataNucleus<sup>17</sup> : Supports the JPA 2.0 `ElementCollection` mapping.

Typically the primary key of the target table will be composed of the parent's primary key, and some unique field in the embeddable object. The embeddable should have a unique field within its parent's collection, but does not need to be unique for the entire class. It could still have a unique id and still use sequencing, or if it has no unique fields, its id could be composed of all of its fields. The embeddable collection object will be different than a typical embeddable object as it will not be stored in the parent's table, but in its own table. Embeddables are strictly privately owned objects, deletion of the parent will cause deletion of the embeddables, and removal from the embeddable collection should cause the embeddable to be deleted. Embeddables cannot be queried directly, and are not independent objects as they have no Id.

## 26.8 Querying

Embeddable objects cannot be queried directly, but they can be queried in the context of their parent. Typically it is best to select the parent, and access the embeddable from the parent. This will ensure the embeddable is registered with the persistence context. If the embeddable is selected in a query, the resulting objects will be detached, and changes will not be tracked.

### Example of querying an embeddable

```
SELECT employee.period from Employee employee where  
employee.period.endDate = :param
```

Embeddables<sup>18</sup> Embeddables<sup>19</sup>

---

14 Chapter 11 on page 25

15 Chapter 10 on page 23

16 Chapter 12 on page 27

17 <http://en.wikibooks.org/wiki/Java%20Persistence%2FDataNucleus>

18 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

19 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FMapping>

## 27 Locking

Locking is perhaps the most ignored persistence consideration. Most applications tend to ignore thinking about concurrency issues during development, and then smush in a locking mechanism before going into production. Considering the large percentage of software projects that fail or are canceled, or never achieve a large user base, perhaps this is logical. However, locking and concurrency is a critical or at least a very important issue for most applications, so probably should be something considered earlier in the development cycle.

If the application will have concurrent writers to the same objects, then a locking strategy is critical so that data corruption can be prevented. There are two strategies for preventing concurrent modification of the same object/row; optimistic<sup>1</sup> and pessimistic<sup>2</sup> locking. Technically there is a third strategy, *ostrich*<sup>3</sup> locking, or no locking, which means put your head in the sand and ignore the issue.

There are various ways to implement both optimistic and pessimistic locking. JPA has support for version optimistic locking, but some JPA providers support other methods of optimistic locking, as well as pessimistic locking.

Locking and concurrency can be a confusing thing to consider, and there are a lot of misconcepts out there. Correctly implementing locking in your application typically involves more than setting some JPA or database configuration option (although that is all many applications that *think*<sup>4</sup> they are using locking do). Locking may also involve application level changes, and ensuring other applications accessing the database also do so correctly for the locking policy being used.

### 27.1 Optimistic Locking

Optimistic locking assumes that the data will not be modified between when you read the data until you write the data. This is the most common style of locking used and recommended in today's persistence solutions. The strategy involves checking that one or more values from the original object read, are still the same when updating it. This verifies that the object has not changed by another user in between the read and the write.

JPA supports using an optimistic locking version field that gets updated on each update. The field can either be numeric or a timestamp value. A numeric value is recommended as a numeric value is more precise, portable, performant and easier to deal with than a timestamp.

---

1 Chapter 27.1 on page 99  
2 Chapter 27.2.7 on page 107  
3 Chapter 27.2.1 on page 104  
4 Chapter 27.1.1 on page 100

The `@Version`<sup>5</sup> annotation or `<version>` element is used to define the optimistic lock version field. The annotation is defined on the version field or property for the object, similar to an Id mapping. The object must contain an attribute to store the version field.

The object's version attribute is automatically updated by the JPA provider, and should not normally be modified by the application. The one exception is if the application reads the object in one transaction, sends the object to a client, and updates/merges the object in another transaction. In this case the application must ensure that the original object version is used, otherwise any changes in between the read and write will not be detected. The `EntityManager merge()` API will always merge the version, so the application is only responsible for this if manually merging.

When a locking contention is detected an `OptimisticLockException`<sup>6</sup> will be thrown. This could be wrapped inside a `RollbackException`, or other exceptions if using JTA, but it should be set as the `cause` of the exception. The application can handle the exception, but should normally report the error to the user, and let them determine what to do.

### Example of Version annotation

```
@Entity
public abstract class Employee{
    @Id
    private long id;
    @Version
    private long version;
    ...
}
```

### Example of Version XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
    <attributes>
        <id name="id"/>
        <version name="version"/>
        ...
    </attributes>
</entity>
```

## 27.1.1 Common Locking Mistakes, Questions and Problems

### *Not sending version to client, only locking on the server*

Probably the most common mistake in locking in general is locking the wrong section of code. This is true no matter what form of locking is used, whether it be optimistic or pessimistic. The basic scenario is:

---

<sup>5</sup> <https://java.sun.com/javaee/5/docs/api/javax/persistence/Version.html>

<sup>6</sup> <https://java.sun.com/javaee/5/docs/api/javax/persistence/OptimisticLockException.html>

1. User requests some data, the server reads the data from the database and sends it to the user in their client (doesn't matter if the client is html, rmi, web service).
2. The user edits the data in the client.
3. The user submits the data back to the server.
4. The server begins a transaction, reads the object, merges the data and commits the transaction.

The issue is that the original data was read in step 1, but the lock was not obtained until step 4, so any changes made to the object in between steps 1 and 4 would not result in a conflict. This means there is little point to using any locking.

A key point is that when using database pessimistic locking or database transaction isolation, this will always be the case, the database locks will only occur in step 4, and any conflicts will not be detected. This is the main reason why using database locking does not scale to web applications, for the locking to be valid, the database transaction must be started at step 1 and not committed until step 4. This means the a live database connection and transaction must be held open while waiting for the web client, as well as locks, since there is no guarantee that the web client will not sit on the data for hours, go to lunch, or disappear off the face of the earth, holding database resources and locking data for all other users can be very undesirable.

For optimistic locking the solution is relatively simple, the object's version must be sent to the client along with the data (or kept in the http session). When the user submits the data back, the original version must be merged into the object read from the database, to ensure that any changes made between step 1 and 4 will be detected.

### *Handling optimistic lock exceptions*

Unfortunately programmers can frequently be too clever for their own good. The first issue that comes up when using optimistic locking is what to do when an `OptimisticLockException` occurs. The typical response of the friendly neighborhood super programmer, is to automatically handle the exception. They will just create a new transaction, refresh the object to reset its version, and merge the data back into the object and re-commit it. Presto problem solved, or is it?

This actually defeats the whole point of locking in the first place. If this is what you desire, you may as well use no locking. Unfortunately, the `OptimisticLockException` should rarely be automatically handled, and you really need to bother the user about the issue. You should report the conflict to the user, and either say "your sorry but an edit conflict occurred and they are going to have to redo their work", or in the best case, refresh the object and present the user with the current data and the data that they submitted and help them merge the two if appropriate.

Some automated merge tools will compare the two conflicting versions of the data and if none of the individual fields conflict, then the data will just be automatically merged without the user's aid. This is what most software version control systems do. Unfortunately the user is typically better able to decide when something is a conflict than the program, just because two versions of the .java file did not change the same line of code does not mean there was no conflict, the first user could have deleted a method that the other user added

a method to reference, and several other possible issues that cause the typically nightly build to break every so often.

### *Paranoid Delusionment*

Locking can prevent most concurrency issues, but be careful of going overboard in over analyzing to death every possible hypothetical occurrence. Sometimes in an concurrent application (or any software application) bad stuff can happen. Users are pretty used to this by now, and I don't think anyone out there thinks computers are perfect.

A good example is a source code control system. Allowing users to overwrite each other changes is a bad thing; so most systems avoid this through versioning the source files. If a user submits changes to a file that originated from a older version than the current version, the source code control system will raise a conflict and make the user merge the two files. This is essentially optimistic locking. But what if one user removes, or renames a method in one file, then another user adds a new method or call in another file to that old method. No source code control system that I know of will detect this issue, it is a conflict and will cause the build to break. The solution to this is to start locking or checking the lock on every file in the system (or at least every possible related file). Similar to using optimistic read locking on every possible related object, or pessimistically locking every possible related object. This could be done, but would probably be very expensive, and more importantly would now raise possible conflicts every time a user checked in, so would be entirely useless.

So, in general be careful of being too paranoid, such that you sacrifice the usability of your system.

### *Other applications accessing same data*

Any form of locking that is going to work requires that all applications accessing the same data follow the same rules. If you use optimistic locking in one application, but no locking in another accessing the same data, they will still conflict. One fake solution is to configure an update trigger to always increment the version value (unless incremented in the update). This will allow the new application to avoid overwriting the old application's changes, but the old application will still be able to overwrite the new application's changes. This still may be better than no locking at all, and perhaps the old application will eventually go away.

One common misconception is that if you use pessimistic locking, instead of adding a version field, you will be ok. Again pessimistic locking requires that all applications accessing the same data use the same form of locking. The old application can still read data (without locking), then update the data after the new application reads, locks, and updates the same data, overwriting its changes.

***Isn't database transaction isolation all I need?***

Possibly, but most likely not. Most databases default to *read committed* transaction isolation. This means that you will never see uncommitted data, but this does not prevent concurrent transactions from overwriting the same data.

1. Transaction A reads row x.
2. Transaction B reads row x.
3. Transaction A writes row x.
4. Transaction B writes row x (and overwrites A's changes).
5. Both commit successfully.

This is the case with *read committed*, but with *serializable* this conflict would not occur. With serializable either Transaction B would lock on the select for B and wait (perhaps a long time) until Transaction A commits. In some databases Transaction A may not wait, but would fail on commit. However, even with serializable isolation the typical web application would still have a conflict. This is because each server request operates in a different database transaction. The web client reads the data in one transaction, then updates it in another transaction. So optimistic locking is really the only viable locking option for the typical web application. Even if the read and write occurs in the same transaction, serializable is normally not the solution because of concurrency implications and deadlock potential.

See Serializable Transaction Isolation<sup>7</sup>

***What happens if I merge an object that was deleted by another user?***

What *should* happen is the merge should trigger an `OptimisticLockException` because the object has a version that is not null and greater than 0, and the object does not exist. But this is probably JPA provider specific, some may re-insert the object (this would occur without locking), or throw a different exception.

If you called `persist` instead of `merge`, then the object would be re-inserted.

***What if my table doesn't have a version column?***

The best solution is probably just to add one. Field locking is another solution, as well as pessimistic locking in some cases.

See Field Locking<sup>8</sup>

***What about relationships?***

See Cascaded Locking<sup>9</sup>

---

<sup>7</sup> Chapter 27.2.9 on page 109

<sup>8</sup> Chapter 27.2.4 on page 105

<sup>9</sup> Chapter 27.2.3 on page 105

### *Can I use a timestamp?*

See Timestamp Locking<sup>10</sup>

### *Do I need a version in each table for inheritance or multiple tables?*

The short answer is no, only in the root table.

See Multiple Versions<sup>11</sup>

## 27.2 Advanced

### 27.2.1 Timestamp Locking

Timestamp version locking is supported by JPA and is configured the same as numeric version locking, except the attribute type will be a `java.sql.Timestamp` or other date/time type. Be cautious in using timestamp locking as timestamps have different levels of precision in different databases, and some database do not store a timestamp's milliseconds, or do not store them precisely. In general timestamp locking is less efficient than numeric version locking, so numeric version locking is recommended.

Timestamp locking is frequently used if the table already has a *last updated* timestamp column, and is also a convenient way to auto update a *last updated* column. The timestamp version value can be more useful than a numeric version, as it includes the relevant information on when the object was last updated.

The timestamp value in timestamp version locking can either come from the database, or from Java (mid-tier). JPA does not allow this to be configured, however some JPA providers may provide this option. Using the database's current timestamp can be very expensive, as it requires a database call to the server.

### 27.2.2 Multiple Versions

An object can only have one version in JPA. Even if the object maps to multiple tables, only the primary table will have the version. If any fields in any of the tables changes, the version will be updated. If you desire multiple versions, you may need to map multiple version attributes in your object and manually maintain the duplicate versions, perhaps through events. Technically there is nothing preventing your from annotating multiple attributes with `@Version`, and potentially some JPA providers may support this (don't forget to sacrifice a chicken).

---

<sup>10</sup> Chapter 27.2.1 on page 104

<sup>11</sup> Chapter 27.2.2 on page 104

### 27.2.3 Cascaded Locking

Locking objects is different than locking rows in the database. An object can be more complex than a simple row; an object can span multiple tables, have inheritance, have relationships, and have dependent objects. So determining when an object has *changed* and needs to update its' version can be more difficult than determining when a row has changed.

JPA does define that when any of the object's tables changes the version is updated. However it is less clear on relationships. If **Basic**, **Embedded**, or a foreign key relationship (**OneToOne**, **ManyToOne**) changes, the version will be updated. But what about **OneToMany**, **ManyToMany**, and a target foreign key **OneToOne**? For changes to these relationships the update to the version may depend on the JPA provider.

What about changes made to dependent objects? JPA does not have a cascade option for locking, and has no direct concept of dependent objects, so this is not an option. Some JPA providers may support this. One way to simulate this is to use write locking<sup>12</sup>. JPA defines the `EntityManager lock()`<sup>13</sup> API. You can define a version only in your root parent objects, and when a child (or relationship) is changed, you can call the lock API with the parent to cause a **WRITE** lock. This will cause the parent version to be updated. You may also be able to automate this through persistence events.

Usage of cascaded locking depends on your application. If in your application you consider one user updating one dependent part of an object, and another user updating another part of the object to be a locking contention, then this is what you want. If your application does not consider this to be a problem, then you do not want cascaded locking. One of the advantages of cascaded locking is you have fewer version fields to maintain, and only the update to the root object needs to check the version. This can make a difference in optimizations such as batch writing, as the dependent objects may not be able to be batched if they have their own version that must be checked.

TopLink<sup>14</sup> / EclipseLink<sup>15</sup> : Support cascaded locking through their `@OptimisticLocking` and `@PrivateOwned` annotations and XML.

### 27.2.4 Field Locking

If you do not have a version field in your table, optimistic field locking is another solution. Field locking involves comparing certain fields in the object when updating. If those fields have changed, then the update will fail. JPA does not support field locking, but some JPA providers do support it.

Field locking can also be used when a finer level of locking is desired. For example if one user changes the object's name and another changes the objects address, you may desire for these updates to not conflict, and only desire optimistic lock errors when users change the

<sup>12</sup> Chapter 27.2.5 on page 106

<sup>13</sup> [https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#lock\(java.lang.Object,%20javax.persistence.LockModeType\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#lock(java.lang.Object,%20javax.persistence.LockModeType))

<sup>14</sup> Chapter 11 on page 25

<sup>15</sup> Chapter 10 on page 23



same fields. You may also only be concerned about conflicts in changes to certain fields, and not desire lock errors from conflicts in the other fields.

Field locking can also be used on legacy schemas, where you cannot add a version column, or to integrate with other applications accessing the same data which are not using optimistic locking (note if the other applications are not also using field locking, you can only detect conflicts in one direction).

There are several types of field locking:

- All fields compared in the update - This can lead to a very big where clause, but will detect any conflicts.
- Selected fields compared in the update - This is useful if conflicts in only certain fields are desired.
- Changed fields compared in the update - This is useful if only changes to the same fields are considered to be conflicts.

If your JPA provider does not support field locking, it is difficult to simulate, as it requires changes to the update SQL. Your JPA provider may allow overriding the update SQL, in which case, **All** or **Selected** field locking may be possible (if you have access to the original values), but **Changed** field locking is more difficult because the update must be dynamic. Another way to simulate field locking is to **flush** you changes, then refresh the object using a separate **EntityManager** and connection and compare the current values with your original object.

When using field locking it is important to keep the original object that was read. If you read the object in one transaction and send it to a client, then update in another, you are not really locking. Any changes made between the read and write will not be detected. You must keep the original object read managed in an **EntityManager** for your locking to have any effect.

TopLink<sup>16</sup> / EclipseLink<sup>17</sup> : Support field locking through their **@OptimisticLocking** annotation and XML.

### 27.2.5 Read and Write Locking

It is sometimes desirable to lock something that you did not change. Normally this is done when making a change to one object, that is based on the state of another object, and you wish to ensure that the other object represents the current state of the database at the point of the commit. This is what serializable transaction isolation gives you, but optimistic read and write locking allow this requirement to be met declaratively and optimistically (and without deadlock, concurrency, and open transaction issues).

JPA supports read and write locks through the **EntityManager.lock()**<sup>18</sup> API. The **LockModeType**<sup>19</sup> argument can either be **READ** or **WRITE**. A **READ** lock will ensure that

---

<sup>16</sup> Chapter 11 on page 25

<sup>17</sup> Chapter 10 on page 23

<sup>18</sup> [https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#lock\(java.lang.Object,%20javax.persistence.LockModeType\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#lock(java.lang.Object,%20javax.persistence.LockModeType))

<sup>19</sup> <https://java.sun.com/javaee/5/docs/api/javax/persistence/LockModeType.html>

the state of the object does not change on commit. A `WRITE` lock will ensure that this transaction conflicts with any other transaction changing or locking the object. Essentially the `READ` lock check the optimistic version field, and the `WRITE` checks and increments it.

### Example of Using the Lock API

```
Employee employee = entityManager.find(Employee.class, id);
employee.setSalary(employee.getManager().getSalary() / 2);
entityManager.lock(employee.getManager(), LockModeType.READ);
```

Write locking can also be used to provide object-level locks. If you desire for a change to a dependent object to conflict with any change to the parent object, or any other of its dependent objects, this can be done through write locking. This can also be used to lock relationships, when you change a `OneToMany` or `ManyToMany` relationship you can also force the parent's version to be incremented.

### Example of Using the Lock API for Cascaded Locks

```
Employee employee = entityManager.find(Employee.class, id);
employee.getAddress().setCity("Ottawa");
entityManager.lock(employee, LockModeType.WRITE);
```

## 27.2.6 No Locking a.k.a *Ostrich Locking*

Conceptually people may scoff and be alarmed at the thought of no locking, but it is probably the most common form of locking in use. Some call it Ostrich locking as the strategy is to stick your head in the sand and ignore the issue. Most prototypes or small applications frequently do not have the requirement or in most cases the need for locking, and handling what to do when a locking contention does occur is beyond the scope of the application, so best to just ignore the issue.

In general it is probably best in JPA to enable optimistic locking always, as it is fairly simple to do, at least in concept, but what does occur on a conflict without any form of locking? Essentially it is last in wins, so if two users edit the same object, at the same time, the last one to commit will have their changes reflected in the database. This is true, at least for users editing the same fields, but if two users edit different fields in the same object, it depends on the JPA implementation. Some JPA providers only update exactly the fields that were changed, where as other update all fields in the object. So in one case the first user's changes would be overridden, but in the second they would not.

## 27.2.7 Pessimistic Locking

Pessimistic locking means acquiring a lock on the object before you begin to edit the object, to ensure that no other users are editing the object. Pessimistic locking is typically implemented through using database row locks, such as through the `SELECT ... FOR`

UPDATE SQL syntax. The data is read and locked, the changes are made and the transaction is committed, releasing the locks.

JPA 1.0 did not support pessimistic locking, but some JPA 1.0 providers do. JPA 2.0<sup>20</sup> supports pessimistic locking. It is also possible to use JPA native SQL queries to issue `SELECT ... FOR UPDATE` and use pessimistic locking. When using pessimistic locking you must ensure that the object is refreshed when it is locked, locking a potentially stale object is of no use. The SQL syntax for pessimistic locking is database specific, and different databases have different syntax and levels of support, so ensure your database properly supports your locking requirements.

EclipseLink<sup>21</sup> (as of 1.2) : Supports JPA 2.0 pessimistic locking.

TopLink<sup>22</sup> / EclipseLink<sup>23</sup> : Support pessimistic locking through the `"eclipselink.pessimistic-lock"` query hint.

The main issues with pessimistic locking is they use database resources, so require a database transaction and connection to be held open for the duration of the edit. This is typically not desirable for interactive web applications. Pessimistic locking can also have concurrency issues and cause deadlocks. The main advantages of pessimistic locking is that once the lock is obtained, it is fairly certain that the edit will be successful. This can be desirable in highly concurrent applications, where optimistic locking may cause too many optimistic locking errors.

There are other ways to implement pessimistic locking, it could be implemented at the application level, or through serializable transaction<sup>24</sup> isolation.

Application level pessimistic locking can be implemented through adding a *locked* field to your object. Before an edit you must update the field to locked (and commit the change). Then you can edit the object, and set the locked field back to false. To avoid conflicts in acquiring the lock, you should also use optimistic locking, to ensure the lock field is not updated to true by another user at the same time.

## 27.2.8 JPA 2.0 Locking

JPA 2.0 adds support for pessimistic locking, as well as other locking options. A lock can be acquired using the `EntityManager.lock()`<sup>25</sup> API, or passing a `LockModeType` to an `EntityManager.find()` or `refresh()` operation, or setting the `lockMode` of a `Query` or `NamedQuery`.

The JPA 2.0 lock modes are defined in the `LockModeType`<sup>26</sup> enum:

---

<sup>20</sup> Chapter 27.2.8 on page 108

<sup>21</sup> Chapter 10 on page 23

<sup>22</sup> Chapter 11 on page 25

<sup>23</sup> Chapter 10 on page 23

<sup>24</sup> Chapter 27.2.9 on page 109

<sup>25</sup> [https://java.sun.com/javaee/6/docs/api/javax/persistence/EntityManager.html#lock\(java.lang.Object,%20javax.persistence.LockModeType\)](https://java.sun.com/javaee/6/docs/api/javax/persistence/EntityManager.html#lock(java.lang.Object,%20javax.persistence.LockModeType))

<sup>26</sup> <https://java.sun.com/javaee/6/docs/api/javax/persistence/LockModeType.html>

- **OPTIMISTIC** (was **READ** in JPA 1.0) - The Entity will have its optimistic lock version checked on commit, to ensure no other transaction updated the object.
- **OPTIMISTIC\_FORCE\_INCREMENT** (was **WRITE** in JPA 1.0) - The Entity will have its optimistic lock version incremented on commit, to ensure no other transaction updated (or **READ** locked) the object.
- **PESSIMISTIC\_READ** - The Entity is locked on the database, prevents any other transaction from acquiring a **PESSIMISTIC\_WRITE** lock.
- **PESSIMISTIC\_WRITE** - The Entity is locked on the database, prevents any other transaction from acquiring a **PESSIMISTIC\_READ** or **PESSIMISTIC\_WRITE** lock.
- **PESSIMISTIC\_FORCE\_INCREMENT** - The Entity is locked on the database, prevents any other transaction from acquiring a **PESSIMISTIC\_READ** or **PESSIMISTIC\_WRITE** lock, and the Entity will have its optimistic lock version incremented on commit. This is unusual as it does both an optimistic and pessimistic lock, normally an application would only use one locking model.
- **NONE** - No lock is acquired, this is the default to any find, refresh or query operation.

JPA 2.0 also adds two new standard query hints. These can be passed to any **Query**, **NamedQuery**, or **find()**, **lock()** or **refresh()** operation.

- `"javax.persistence.lock.timeout"` - Number of milliseconds to wait on the lock before giving up and throwing a **PessimisticLockException**.
- `"javax.persistence.lock.scope"` - The valid scopes are defined in **PessimisticLockScope**<sup>27</sup>, either **NORMAL** or **EXTENDED**. **EXTENDED** will also lock the object's owned join tables and element collection tables.

### 27.2.9 Serializable Transaction Isolation

Serializable transaction isolation guarantees that anything read in the transaction will not be updated by any other user. Through using serializable transaction isolation and ensuring the data being edited is read in the same transaction, you can achieve pessimistic locking. It is important to ensure the objects are refreshed from the database in the transaction, as editing cached or potentially stale data defeats the point of locking.

Serializable transaction isolation can typically be enabled on the database, some databases even have this as the default. It can also be set on the **JDBC Connection**, or through native **SQL**, but this is database specific and different databases have different levels of support. The main issues with serializable transaction isolation are the same as using **SELECT ... FOR UPDATE** (see above for the gory details), in addition everything read is locked, so you cannot decide to only lock certain objects at certain times, but lock everything all the time. This can be a major concurrency issue for transactions with common read-only data, and can lead to deadlocks.

How database implement serializable transaction isolation differs between databases. Some databases (such as Oracle) can perform serializable transaction isolation in more of an optimistic sense, than the typically pessimistic implementation. Instead of each transaction requiring locks on all the data as it is read, the row versions are not checked until the

<sup>27</sup> <https://java.sun.com/javase/6/docs/api/javax/persistence/PessimisticLockScope.html>

transaction is committed, if any of the data changed an exception is thrown and the transaction is not allowed to commit.

Locking <sup>28</sup>

---

<sup>28</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FMapping>

## 28 Basics

A basic attribute is one where the attribute class is a simple type such as `String`, `Number`, `Date` or a primitive. A basic attribute's value can map directly to the column value in the database. The following table summarizes the basic types and the database types they map to.

Java type	Database type
<code>String</code> ( <code>char</code> , <code>char[]</code> )	<code>VARCHAR</code> ( <code>CHAR</code> , <code>VARCHAR2</code> , <code>CLOB</code> , <code>TEXT</code> )
<code>Number</code> ( <code>BigDecimal</code> , <code>BigInteger</code> , <code>Integer</code> , <code>Double</code> , <code>Long</code> , <code>Float</code> , <code>Short</code> , <code>Byte</code> ) <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> , <code>short</code> , <code>byte</code>	<code>NUMERIC</code> ( <code>NUMBER</code> , <code>INT</code> , <code>LONG</code> , <code>FLOAT</code> , <code>DOUBLE</code> ) <code>NUMERIC</code> ( <code>NUMBER</code> , <code>INT</code> , <code>LONG</code> , <code>FLOAT</code> , <code>DOUBLE</code> )
<code>byte[]</code> <code>boolean</code> ( <code>Boolean</code> )	<code>VARBINARY</code> ( <code>BINARY</code> , <code>BLOB</code> ) <code>BOOLEAN</code> ( <code>BIT</code> , <code>SMALLINT</code> , <code>INT</code> , <code>NUMBER</code> )
<code>java.util.Date</code> <code>java.sql.Date</code> <code>java.sql.Time</code> <code>java.sql.Timestamp</code> <code>java.util.Calendar</code> <code>java.lang.Enum</code> <code>java.util.Serializable</code>	<code>TIMESTAMP</code> ( <code>DATE</code> , <code>DATETIME</code> ) <code>DATE</code> ( <code>TIMESTAMP</code> , <code>DATETIME</code> ) <code>TIME</code> ( <code>TIMESTAMP</code> , <code>DATETIME</code> ) <code>TIMESTAMP</code> ( <code>DATETIME</code> , <code>DATE</code> ) <code>TIMESTAMP</code> ( <code>DATETIME</code> , <code>DATE</code> ) <code>NUMERIC</code> ( <code>VARCHAR</code> , <code>CHAR</code> ) <code>VARBINARY</code> ( <code>BINARY</code> , <code>BLOB</code> )

In JPA a basic attribute is mapped through the `@Basic`<sup>1</sup> annotation or the `<basic>` element. The types and conversions supported depend on the JPA implementation and database platform. Some JPA implementations may support conversion between many different data-types or additional types, or have extended type conversion support, see the advanced<sup>2</sup> section for more details. Any basic attribute using a type that does not map directly to a database type can be serialized to a binary database type.

The easiest way to map a basic attribute in JPA is to do nothing. Any attributes that have no other annotations and do not reference other entities will be automatically mapped as basic, and even serialized if not a basic type. The column name for the attribute will be defaulted, named the same as the attribute name, as uppercase. Sometimes auto-mapping can be unexpected if you have an attribute in your class that you did not intend to have

---

<sup>1</sup> <https://java.sun.com/javase/5/docs/api/javax/persistence/Basic.html>

<sup>2</sup> Chapter 29 on page 115

persisted. You must mark any such non-persistent fields using the `@Transient`<sup>3</sup> annotation or `<transient>` element.

Although auto-mapping makes rapid prototyping easy, you typically reach a point where you want control over your database schema. To specify the column name for a basic attribute the `@Column`<sup>4</sup> annotation or `<column>` element is used. The column annotation also allows for other information to be specified such as the database type, size, and some constraints.

### 28.0.10 Example of basic mapping annotations

```
@Entity
public class Employee {
    // Id mappings are also basic mappings.
    @Id
    @Column(name="ID")
    private long id;
    @Basic
    @Column(name="F_NAME")
    private String firstName;
    // The @Basic is not required in general because it is the default.
    @Column(name="L_NAME")
    private String lastName;
    // Any un-mapped field will be automatically mapped as basic and
    column name defaulted.
    private BigDecimal salary;
    // Non-persistent fields must be marked as transient.
    @Transient
    private EmployeeService service;
    ...
}
```

### 28.0.11 Example of basic mapping XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id">
      <column name="ID"/>
    </id>
    <basic name="firstName">
      <column name="F_NAME"/>
    </basic>
    <basic name="lastName">
      <column name="L_NAME"/>
    </basic>
    <transient name="service"/>
  </attributes>
</entity>
```

---

3 <https://java.sun.com/javase/5/docs/api/javax/persistence/Transient.html>

4 <https://java.sun.com/javase/5/docs/api/javax/persistence/Column.html>

## 28.1 Common Problems

### 28.1.1 *Translating Values*

See Conversion<sup>5</sup>

### 28.1.2 *Truncated Data*

A common issue is that data, such as Strings, written from the object are truncated when read back from the database. This is normally caused by the column length not being large enough to handle the object's data. In Java there is no maximum size for a String, but in a database `VARCHAR` field, there is a maximum size. You must ensure that the length you set in your column when you create the table is large enough to handle any object value. For very large Strings `CLOBs` can be used, but in general `CLOBs` should not be over used, as they are less efficient than a `VARCHAR`.

If you use JPA to generate your database schema, you can set the column length through the `Column` annotation or element, see Column Definition and Schema Generation<sup>6</sup>.

### 28.1.3 *How to map timestamp with timezones?*

See Timezones<sup>7</sup>

### 28.1.4 *How to map XML data-types?*

See Custom Types<sup>8</sup>

### 28.1.5 *How to map Struct and Array types?*

See Custom Types<sup>9</sup>

### 28.1.6 *How to map custom database types?*

See Custom Types<sup>10</sup>

---

5 Chapter 29.8 on page 122  
6 Chapter 29.6 on page 120  
7 Chapter 29.1.4 on page 116  
8 Chapter 29.9 on page 122  
9 Chapter 29.9 on page 122  
10 Chapter 29.9 on page 122



**28.1.7 *How to exclude fields from INSERT or UPDATE statements, or default values in triggers?***

See Insertable, Updatable<sup>11</sup>

---

<sup>11</sup> Chapter 29.7 on page 121

## 29 Advanced

### 29.1 Temporal, Dates, Times, Timestamps and Calendars

Dates, times, and timestamps are common types both in the database and in Java, so in theory mappings these types should be simple, right? Well sometimes this is the case and just a normal `Basic` mapping can be used, however sometimes it becomes more complex.

Some databases do not have `DATE` and `TIME` types, only `TIMESTAMP` fields, however some do have separate types, and some just have `DATE` and `TIMESTAMP`. Originally in Java 1.0, Java only had a `java.util.Date` type, which was both a date, time and milliseconds. In Java 1.1 this was expanded to support the common database types with `java.sql.Date`, `java.sql.Time`, and `java.sql.Timestamp`, then to support internationalization Java created the `java.util.Calendar` type and virtually deprecated (almost all of the methods) the old date types (which JDBC still uses).

If you map a Java `java.sql.Date` type to a database `DATE`, this is just a basic mapping and you should not have any issues (ignore Oracle's `DATE` type that is/was a timestamp for now). You can also map `java.sql.Time` to `TIME`, and `java.sql.Timestamp` to `TIMESTAMP`. However if you have a `java.util.Date` or `java.util.Calendar` in Java and wish to map it to a `DATE` or `TIME`, you may need to indicate that the JPA provider perform some sort of conversion for this. In JPA the `@Temporal`<sup>1</sup> annotation or `<temporal>` element is used to map this. You can indicate that just the `DATE` or `TIME` portion of the date/time value be stored to the database. You could also use `Temporal` to map a `java.sql.Date` to a `TIMESTAMP` field, or any other such conversion.

#### 29.1.1 Example of temporal annotation

```
@Entity
public class Employee {
    ...
    @Basic
    @Temporal(DATE)
    private Calendar startDate;
    ...
}
```

#### 29.1.2 Example of temporal XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
```

---

<sup>1</sup> <https://java.sun.com/javase/5/docs/api/javax/persistence/Temporal.html>

```
<attributes>
  ...
  <basic name="startDate">
    <temporal>DATE</temporal>
  </basic>
</attributes>
</entity>
```

### 29.1.3 Milliseconds

The precision of milliseconds is different for different temporal classes and database types, and on different databases. The `java.util.Date` and `Calendar` classes support milliseconds. The `java.sql.Date` and `java.sql.Time` classes do not support milliseconds. The `java.sql.Timestamp` class supports nanoseconds.

On many databases the `TIMESTAMP` type supports milliseconds. On Oracle prior to Oracle 9, there was only a `DATE` type, which was a date and a time, but had no milliseconds. Oracle 9 added a `TIMESTAMP` type that has milliseconds (and nanoseconds), and now treats the old `DATE` type as only a date, so be careful using it as a timestamp. MySQL has `DATE`, `TIME` and `DATETIME` types. DB2 has a `DATE`, `TIME` and `TIMESTAMP` types, the `TIMESTAMP` supports microseconds. Sybase and SQL Server just have a `DATETIME` type which has milliseconds, but at least on some versions has precision issues, it seems to store an estimate of the milliseconds, not the exact value.

If you use timestamp version locking you need to be very careful of your milliseconds precision. Ensure your database supports milliseconds precisely otherwise you may have issues, especially if the value is assigned in Java, then differs what gets stored on the database, which will cause the next update to fail for the same object.

In general I would not recommend using a timestamp and as primary key or for version locking. There are too many database compatibility issues, as well as the obvious issue of not supporting two operations in the same millisecond.

### 29.1.4 Timezones

Temporals become a lot more complex when you start to consider time zones, internationalization, eras, locals, day-light savings time, etc. In Java only `Calendar` supports time zones. Normally a `Calendar` is assumed to be in the local time zone, and is stored and retrieved from the database with that assumption. If you then read that same `Calendar` on another computer in another time zone, the question is if you will have the same `Calendar` or will you have the `Calendar` of what the original time would have been in the new time zone? It depends on if the `Calendar` is stored as the GMT time, or the local time, and if the time zone was stored in the database.

Some databases support time zones, but most database types do not store the time zone. Oracle has two special types for timestamps with time zones, `TIMESTAMP TZ` (time zone is stored) and `TIMESTAMP LTZ` (local time zone is used). Some JPA providers may have extended support for storing `Calendar` objects and time zones.

TopLink<sup>2</sup>, EclipseLink<sup>3</sup> : Support the Oracle `TIMESTAMPZ` and `TIMESTAMPZ` types using the `@TypeConverter` annotation and XML.

## Forum Posts

- Investigation of storing timezones in MySQL<sup>4</sup>

## 29.2 Enums

Java `Enums`<sup>5</sup> are typically used as constants in an object model. For example an `Employee` may have a `gender` of enum type `Gender` (`MALE`, `FEMALE`).

By default in JPA an attribute of type Enum will be stored as a `Basic` to the database, using the integer Enum values as codes (i.e. 0, 1). JPA also defines an `@Enumerated`<sup>6</sup> annotation and `<enumerated>` element (on a `<basic>`) to define an Enum attribute. This can be used to store the Enum as the `STRING` value of its name (i.e. `"MALE"`, `"FEMALE"`).

For translating Enum types to values other than the integer or String name, such as character constants, see Translating Values<sup>7</sup>.

### 29.2.1 Example of enumerated annotation

```
public enum Gender {
    MALE,
    FEMALE
}

@Entity
public class Employee {
    ...
    @Basic
    @Enumerated(EnumType.STRING)
    private Gender gender;
    ...
}
```

### 29.2.2 Example of enumerated XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    ...
    <basic name="gender">
      <enumerated>STRING</enumerated>
    </basic>
  </attributes>
</entity>
```

---

2 Chapter 11 on page 25  
 3 Chapter 10 on page 23  
 4 <http://old.nabble.com/MySQL%27s-datetime-and-time-zones--td21006801.html>  
 5 <https://java.sun.com/java/5/docs/api/java/lang/Enum.html>  
 6 <https://java.sun.com/javase/5/docs/api/javax/persistence/Enumerated.html>  
 7 Chapter 29.1.4 on page 116

```
</attributes>
</entity>
```

## 29.3 LOBs, BLOBs, CLOBs and Serialization

A LOB is a Large Object, such as a BLOB (Binary LOB), or a CLOB (Character LOB). It is a database type that can store a large binary or string value, as the normal `VARCHAR` or `VARBINARY` types typically have size limitations. A LOB is often stored as a locator in the database table, with the actual data stored outside of the table. In Java a CLOB will normally map to a `String`, and a BLOB will normally map to a `byte[]`, although a BLOB may also represent some serialized object.

By default in JPA any `Serializable` attribute that is not a relationship or a basic type (`String`, `Number`, temporal, primitive), will be serialized to a BLOB field.

JPA defines the `@Lob`<sup>8</sup> annotation and `<lob>` element (on a `<basic>`) to define that an attribute maps to a LOB type in the database. The annotation is just a hint to the JPA implementation that this attribute will be stored in a LOB, as LOBs may need to be persisted specially. Sometimes just mapping the LOB as a normal `Basic` will work fine as well.

Various databases and JDBC drivers have various limits for LOB sizes. Some JDBC drivers have issues beyond 4k, 32k or 1meg. The Oracle thin JDBC drivers had a 4k limitation in some versions for binding LOB data. Oracle provided a workaround for this limitation, which some JPA providers support. For reading LOBs, some JDBC drivers prefer using streams, some JPA providers also support this option.

Typically the entire LOB will be read and written for the attribute. For very large LOBs reading the value always, or reading the entire value may not be desired. The fetch type of the `Basic` could be set to `LAZY` to avoid reading a LOB unless accessed. Support for `LAZY` fetching on `Basic` is optional in JPA, so some JPA providers may not support it. A workaround, which is often a good idea in general given the large performance cost of LOBs, is to store the LOB in a separate table and class and define a `OneToOne` to the LOB object instead of a `Basic`. If the entire LOB is never desired to be read, then it should not be mapped. It is best to use direct JDBC to access and stream the LOB in this case. It may be possible to map the LOB to a `java.sql.Blob`/`java.sql.Clob` in your object to avoid reading the entire LOB, but these require a live connection, so may have issues with detached objects.

### 29.3.1 Example of lob annotation

```
@Entity
public class Employee {
    ...
    @Basic(fetch=FetchType.LAZY)
    @Lob
    private Image picture;
    ...
}
```

---

8 <https://java.sun.com/javase/5/docs/api/javax/persistence/Lob.html>

### 29.3.2 Example of lob XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    ...
    <basic name="picture" fetch="LAZY">
      <lob/>
    </basic>
  </attributes>
</entity>
```

## 29.4 Lazy Fetching

The `fetch` attribute can be set on a `Basic` mapping to use `LAZY` fetching. By default all `Basic` mappings are `EAGER`, which means the column is selected whenever the object is selected. By setting the `fetch` to `LAZY`, the column will not be selected with the object. If the attribute is accessed, then the attribute value will be selected in a separate database select. Support for `LAZY` is an optional feature of JPA, so some JPA providers may not support it. Typically support for lazy on basics will require some form of byte code weaving, or dynamic byte code generation, which may have issues in certain environments or JVMs, or may require preprocessing your application's persistence unit jar.

Only attributes that are rarely accessed should be marked lazy, as accessing the attribute causes a separate database select, which can hurt performance. This is especially true if a large number of objects is queried. The original query will require one database select, but if each object's lazy attribute is accessed, this will require `n` database selects, which can be a major performance issue.

Using lazy fetching on basics is similar to the concept of fetch groups. Lazy basics is basically support for a single default fetch group. Some JPA providers support fetch groups in general, which allow more sophisticated control over what attributes are fetched per query.

TopLink<sup>9</sup>, EclipseLink<sup>10</sup> : Support lazy basics and fetch groups. Fetch groups can be configured through the EclipseLink API using the `FetchGroup` class.

## 29.5 Optional

A `Basic` attribute can be `optional` if its value is allowed to be null. By default everything is assumed to be optional, except for an `Id`, which can not be optional. `Optional` is basically only a hint that applies to database schema generation, if the persistence provider is configured to generate the schema. It adds a `NOT NULL` constraint to the column if `false`. Some JPA providers also perform validation of the object for optional attributes, and will throw a validation error before writing to the database, but this is not required by the JPA specification. `Optional` is defined through the `optional` attribute of the `Basic` annotation or element.

---

<sup>9</sup> Chapter 11 on page 25

<sup>10</sup> Chapter 10 on page 23

## 29.6 Column Definition and Schema Generation

There are various attributes on the `Column`<sup>11</sup> annotation and element for database schema generation. If you do not use JPA to generate your schema you can ignore these. Many JPA providers do provide the feature of auto generation of the database schema. By default the Java types of the object's attributes are mapped to their corresponding database type for the database platform you are using. You may require configuring your database platform with your provider (such as a `persistence.xml` property) to allow schema generation for your database, as many database use different type names.

The `columnDefinition` attribute of `Column` can be used to override the default database type used, or enhance the type definition with constraints or other such DDL. The `length`, `scale` and `precision` can also be set to override defaults. Since the defaults for the `length` are just defaults, it is normally a good idea to set these to be correct for your data model's expected data, to avoid data truncation. The `unique` attribute can be used to define a unique constraint on the column, most JPA providers will automatically define primary key and foreign key constraints based on the `Id` and relationship mappings.

JPA does not define any options to define an index. Some JPA providers may provide extensions for this. You can also create your own indexes through native queries

### 29.6.1 Example of column annotations

```
@Entity
public class Employee {
    @Id
    @Column(name="ID")
    private long id;
    @Column(name="SSN" unique=true optional=false)
    private long ssn;
    @Column(name="F_NAME" length=100)
    private String firstName;
    @Column(name="L_NAME" length=200)
    private String lastName;
    @Column(name="SALARY" scale=10 precision=2)
    private BigDecimal salary;
    @Column(name="S_TIME" columnDefinition="TIMESTAMPZ")
    private Calendar startTime;
    @Column(name="E_TIME" columnDefinition="TIMESTAMPZ")
    private Calendar endTime;
    ...
}
```

### 29.6.2 Example of column XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id">
      <column name="ID"/>
    </id>
    <basic name="ssn">
```

---

<sup>11</sup> <https://java.sun.com/javase/5/docs/api/javax/persistence/Column.html>

```
        <column name="SSN" unique="true" optional="false"/>
    </basic>
    <basic name="firstName">
        <column name="F_NAME" length="100"/>
    </basic>
    <basic name="lastName">
        <column name="L_NAME" length="200"/>
    </basic>
    <basic name="startTime">
        <column name="S_TIME" columnDefinition="TIMESTAMP TZ"/>
    </basic>
    <basic name="endTime">
        <column name="E_TIME" columnDefinition="TIMESTAMP TZ"/>
    </basic>
</attributes>
</entity>
```

If using `BigDecimal` with `Postgresql`, JPA maps salary to a table column of type `NUMERIC(38,0)`. You can adjust scale and precision for `BigDecimal` within the `@Column` annotation.

```
@Column(precision=8, scale=2)
private BigDecimal salary;
```

## 29.7 Insertable, Updatable / Read Only Fields / Returning

The `Column` annotation and XML element defines `insertable` and `updatable` options. These allow for this column, or foreign key field to be omitted from the SQL `INSERT` or `UPDATE` statement. These can be used if constraints on the table prevent insert or update operations. They can also be used if multiple attributes map to the same database column, such as with a foreign key field through a `ManyToOne` and `Id` or `Basic` mapping. Setting both `insertable` and `updatable` to `false`, effectively mark the attribute as read-only.

`insertable` and `updatable` can also be used in the database table defaults, or auto assigns values to the column on insert or update. Be careful in doing this though, as this means that the object's values will be out of synch with the database, unless it is refreshed. For `IDENTITY` or auto assigned id columns a `GeneratedValue` should normally be used, instead of setting `insertable` to `false`. Some JPA providers also support returning auto assigned fields values from the database after insert or update operations. The cost of refreshing or returning fields back into the object can effect performance, so it is normally better to initialize field values in the object model, not in the database.

`TopLink`<sup>12</sup>, `EclipseLink`<sup>13</sup> : Support returning insert and update values back into the object using the `ReturnInsert` and `ReturnUpdate` annotations and XML elements.

---

<sup>12</sup> Chapter 11 on page 25

<sup>13</sup> Chapter 10 on page 23



## 29.8 Conversion

A common problem in storing values to the database is that the value desired in Java differs from the value used in the database. Common examples include using a `boolean` in Java and a 0, 1 or a 'T', 'F' in the database. Other examples are using a `String` in Java and a `DATE` in the database.

One way to accomplish this is to translate the data through property get/set methods.

```
@Entity
public class Employee {
    ...
    private boolean isActive;
    ...
    @Transient
    public boolean getIsActive() {
        return isActive;
    }
    public void setIsActive(boolean isActive) {
        this.isActive = isActive;
    }
    @Basic
    private String getIsActiveValue() {
        if (isActive) {
            return "T";
        } else {
            return "F";
        }
    }
    private void setIsActiveValue(String isActive) {
        this.isActive = "T".equals(isActive);
    }
}
```

Also for translating date/times see, Temporals<sup>14</sup>.

As well some JPA providers have support for this.

TopLink<sup>15</sup>, EclipseLink<sup>16</sup> : Support translation using the `@Convert`, `@Converter`, `@ObjectTypeConverter` and `@TypeConverter` annotations and XML.

## 29.9 Custom Types

JPA defines support for most common database types, however some databases and JDBC driver have additional types that may require additional support.

Some custom database types include:

- `TIMESTAMP`PTZ, `TIMESTAMP`PLTZ (Oracle)
- `TIMESTAMP` WITH `TIMEZONE` (Postgres)
- `XMLTYPE` (Oracle)
- `XML` (DB2)

---

<sup>14</sup> Chapter 29.1 on page 115

<sup>15</sup> Chapter 11 on page 25

<sup>16</sup> Chapter 10 on page 23

- NCHAR, NVARCHAR, NCLOB (Oracle)
- Struct (STRUCT Oracle)
- Array (VARRAY Oracle)
- BINARY\_INTEGER, DEC, INT, NATURAL, NATURALN, BOOLEAN (Oracle)
- POSITIVE, POSITIVEN, SIGNTYPE, PLS\_INTEGER (Oracle)
- RECORD, TABLE (Oracle)
- SDO\_GEOMETRY (Oracle)
- LOBs (Oracle thin driver)

To handle persistence to custom database types either custom hooks are required in your JPA provider, or you need to mix raw JDBC code with your JPA objects. Some JPA provider provide custom support for many custom database types, some also provide custom hooks for adding your own JDBC code to support a custom database type.

TopLink<sup>17</sup>, EclipseLink<sup>18</sup> : Support several custom database types including, TIMESTAMPTZ, TIMESTAMPLTZ, XMLTYPE, NCHAR, NVARCHAR, NCLOB, object-relational Struct and Array types, PLSQL types, SDO\_GEOMETRY and LOBs.

Basic Attributes<sup>19</sup> Basic Attributes<sup>20</sup>

---

17 Chapter 11 on page 25

18 Chapter 10 on page 23

19 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

20 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FMapping>



## 30 Relationships

A relationship is a reference from one object to another. In Java, relationships are defined through object references (pointers) from a source object to the target object. Technically, in Java there is no difference between a relationship to another object and a "relationship" to a data attribute such as a `String` or `Date` (primitives are different), as both are pointers; however, logically and for the sake of persistence, data attributes are considered part of the object, and references to other persistent objects are considered relationships.

In a relational database relationships are defined through foreign keys. The source row contains the primary key of the target row to define the relationship (and sometimes the inverse). A query must be performed to read the target objects of the relationship using the foreign key and primary key information.

In Java, if a relationship is to a collection of other objects, a `Collection` or array type is used in Java to hold the contents of the relationship. In a relational database, collection relations are either defined by the target objects having a foreign key back to the source object's primary key, or by having an intermediate join table to store the relationship (both objects' primary keys).

All relationships in Java and JPA are unidirectional, in that if a source object references a target object there is no guarantee that the target object also has a relationship to the source object. This is different than a relational database, in which relationships are defined through foreign keys and querying such that the inverse query always exists.

### 30.0.1 JPA Relationship Types

- `OneToOne`<sup>1</sup> - A unique reference from one object to another, inverse of a `OneToOne`.
- `ManyToOne`<sup>2</sup> - A reference from one object to another, inverse of a `OneToMany`.
- `OneToMany`<sup>3</sup> - A `Collection` or `Map` of objects, inverse of a `ManyToOne`.
- `ManyToMany`<sup>4</sup> - A `Collection` or `Map` of objects, inverse of a `ManyToMany`.
- `Embedded`<sup>5</sup> - A reference to a object that shares the same table of the parent.
- `ElementCollection`<sup>6</sup> - JPA 2.0, a `Collection` or `Map` of `Basic` or `Embedable` objects, stored in a separate table.

This covers the majority of types of relationships that exist in most object models. Each type of relationship also covers multiple different implementations, such as `OneToMany`

---

1 Chapter 31.7 on page 150  
2 Chapter 33.2.2 on page 164  
3 Chapter 35.1 on page 169  
4 Chapter 37.1.2 on page 178  
5 Chapter 24.2.4 on page 87  
6 Chapter 39.1.2 on page 185

allowing either a join table, or foreign key in the target, and collection mappings also allow `Collection` types and `Map` types. There are also other possible complex relationship types, see Advanced Relationships<sup>7</sup>.

## 30.1 Lazy Fetching

The cost of retrieving and building an object's relationships, far exceeds the cost of selecting the object. This is especially true for relationships such as `manager` or `managedEmployees` such that if any employee were selected it would trigger the loading of every employee through the relationship hierarchy. Obviously this is a bad thing, and yet having relationships in objects is very desirable.

The solution to this issue is lazy fetching (lazy loading). Lazy fetching allows the fetching of a relationship to be deferred until it is accessed. This is important not only to avoid the database access, but also to avoid the cost of building the objects if they are not needed.

In JPA lazy fetching can be set on any relationship using the `fetch` attribute. The `fetch` can be set to either `LAZY` or `EAGER` as defined in the `FetchType`<sup>8</sup> enum. The default fetch type is `LAZY` for all relationships except for `OneToOne` and `ManyToOne`, but in general it is a good idea to make every relationship `LAZY`. The `EAGER` default for `OneToOne` and `ManyToOne` is for implementation reasons (more difficult to implement), not because it is a good idea. Technically in JPA `LAZY` is just a hint, and a JPA provider is not required to support it, however in reality all main JPA providers support it, and they would be pretty useless if they did not.

### 30.1.1 Example of a lazy one to one relationship annotations

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="ADDR_ID")
    private Address address;
    ...
}
```

### 30.1.2 Example of a lazy one to one relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <one-to-one name="address" fetch="LAZY">
      <join-column name="ADDR_ID"/>
    </one-to-one>
  </attributes>
</entity>
```

---

7 Chapter 31.1 on page 139

8 <https://java.sun.com/javase/5/docs/api/javax/persistence/FetchType.html>

```
</attributes>
</entity>
```

### 30.1.3 Magic

Lazy fetching normally involves some sort of *magic* in the JPA provider to transparently fault in the relationships as they are accessed. The typical magic for collection relationships is for the JPA provider to set the relationships to its own `Collection`, `List`, `Set` or `Map` implementation. When any (or most) method is accessed on this collection proxy, it loads the real collection and forwards the method. This is why JPA requires that all collection relationships use one of the collection interfaces (although some JPA providers support collection implementations too).

For `OneToOne` and `ManyToOne` relationships the magic normally involves some sort of byte code manipulation of the entity class, or creation of a subclass. This allows the access to the field or get/set methods to be intercepted, and for the relationships to be first retrieved before allowing access to the value. Some JPA providers use different methods, such as wrapping the reference in a proxy object, although this can have issues with `null` values and primitive methods. To perform the byte code magic normally an agent or post-processor is required. Ensure that you correctly use your providers agent or post-processor otherwise lazy may not work. You may also notice additional variables when in a debugger, but in general debugging will still work as normal.

### 30.1.4 Basics

A `Basic` attribute can also be made `LAZY`, but this is normally a different mechanism than lazy relationships, and should normally be avoided unless the attribute is rarely accessed.

See Basic Attributes : Lazy Fetching<sup>9</sup>.

### 30.1.5 Serialization, and Detaching

A major issue with lazy relationships, is ensuring that the relationship is still available after the object has been detached, or serialized. For most JPA providers, after serialization any lazy relationship that was not instantiated will be broken, and either throw an error when accessed, or return `null`.

The naive solution is to make every relationship eager. Serialization suffers from the same issue as persistence, in that you can very easily serialize your entire database if you have no lazy relationships. So lazy relationships are just as necessary for serialization, as they are for database access, however you need to ensure you have everything you will need after serialization instantiated upfront. You may mark only the relationships that you think you will need after serialization as `EAGER`, this will work, however there are probably many cases when you do not need these relationships.

---

<sup>9</sup> Chapter 29.4 on page 119

A second solution is to access any relationship you will need before returning the object for serialization. This has the advantage of being use case specific, so different use cases can instantiate different relationships. For collection relationships sending `size()` is normally the best way to ensure a lazy relationship is instantiated. For **OneToOne** and **ManyToOne** relationships, normally just accessing the relationship is enough (i.e. `employee.getAddress()`), although for some JPA providers that use proxies you may need to send the object a message (i.e. `employee.getAddress().hashCode()`).

A third solution is to use the JPQL **JOIN FETCH** for the relationship when querying the objects. A join fetch will normally ensure the relationship has been instantiated. Some caution should be used with join fetch however, as it can become inefficient if used on collection relationships, especially multiple collection relationships as it requires an  $n^2$  join on the database.

Some JPA providers may also provide certain query hints, or other such serialization options.

The same issue can occur without serialization, if a detached object is accessed after the end of the transaction. Some JPA providers allow lazy relationship access after the end of the transaction, or after the **EntityManager** has been closed, however some do not. If your JPA provider does not, then you may require that you ensure you have instantiated all the lazy relationships that you will need before ending the transaction.

### 30.1.6 Eager Join Fetching

One common misconception is that **EAGER** means that the relationship should be join fetched, i.e. retrieved in the same SQL **SELECT** statement as the source object. Some JPA providers do implement eager this way. However, just because something is desired to be loaded, does not mean that it should be join fetched. Consider **Employee - Phone**, a **Phone**'s employee reference is made **EAGER** as the employee is almost always loaded before the phone. However when loading the phone, you do not want to join the employee, the employee has already been read and is already in the cache or persistence context. Also just because you want two collection relationships loaded, does not mean you want them join fetch which would result in a very inefficient join that would return  $n^2$  data.

Join fetching is something that JPA currently only provides through JPQL, which is normally the correct place for it, as each use case has different relationship requirements. Some JPA providers also provide a join fetch option at the mapping level to always join fetch a relationship, but this is normally not the same thing as **EAGER**. Join fetching is not normally the most efficient way to load a relationship anyway, normally batch reading a relationship is much more efficient when supported by your JPA provider.

See Join Fetching<sup>10</sup>

See Batch Reading<sup>11</sup>

---

<sup>10</sup> Chapter 31.3 on page 145

<sup>11</sup> Chapter 31.4 on page 147

## 30.2 Cascading

Relationship mappings have a `cascade` option that allows the relationship to be cascaded for common operations. `cascade` is normally used to model dependent relationships, such as `Order -> OrderLine`. Cascading the `orderLines` relationship allows for the `Order`'s `-> OrderLines` to be persisted, removed, merged along with their parent.

The following operations can be cascaded, as defined in the `CascadeType`<sup>12</sup> enum:

- **PERSIST** - Cascaded the `EntityManager.persist()` operation. If `persist()` is called on the parent, and the child is also new, it will also be persisted. If it is existing, nothing will occur, although calling `persist()` on an existing object will still cascade the persist operation to its dependents. If you persist an object, and it is related to a new object, and the relationship does not cascade persist, then an exception will occur. This may require that you first call `persist` on the related object before relating it to the parent. General it may seem odd, or be desirable to always cascade the persist operation, if a new object is related to another object, then it should probably be persisted. There is most likely not a major issue with always cascading persist on every relationship, although it may have an impact on performance. Calling `persist` on a related object is not required, on commit any related object whose relationship is cascade persist will automatically be persisted. The advantage of calling `persist` up front is that any generated ids will (unless using identity) be assigned, and the `prePersist` event will be raised.
- **REMOVE** - Cascaded the `EntityManager.remove()` operation. If `remove()` is called on the parent then the child will also be removed. This should only be used for dependent relationships. Note that only the `remove()` operation is cascaded, if you remove a dependent object from a `OneToMany` collection it will not be deleted, JPA requires that you explicitly call `remove()` on it. Some JPA providers may support an option to have objects removed from dependent collection deleted, JPA 2.0 also defines an option for this.
- **MERGE** - Cascaded the `EntityManager.merge()` operation. If `merge()` is called on the parent, then the child will also be merged. This should normally be used for dependent relationships. Note that this only effects the cascading of the merge, the relationship reference itself will always be merged. This can be a major issue if you use `transient` variables to limit serialization, you may need to manually merge, or reset transient relationships in this case. Some JPA providers provide additional `merge` operations.
- **REFRESH** - Cascaded the `EntityManager.refresh()` operation. If `refresh()` is called on the parent then the child will also be refreshed. This should normally be used for dependent relationships. Be careful enabling this for all relationships, as it could cause changes made to other objects to be reset.
- **ALL** - Cascaded all the above operations.

### 30.2.1 Example of a cascaded one to one relationship annotations

```
@Entity
public class Employee {
    @Id
```

<sup>12</sup> <https://java.sun.com/javase/5/docs/api/javax/persistence/CascadeType.html>



```
private long id;
...
@OneToOne(cascade={CascadeType.ALL})
@JoinColumn(name="ADDR_ID")
private Address address;
...
}
```

### 30.2.2 Example of a cascaded one to one relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <one-to-one name="address">
      <join-column name="ADDR_ID"/>
      <cascade>
        <cascade-all/>
      </cascade>
    </one-to-one>
  </attributes>
</entity>
```

## 30.3 Orphan Removal (JPA 2.0)

Cascading of the `remove` operation only occurs when the `remove` is called on the object. This is not normally what is desired on a dependent relationship. If the related objects cannot exist without the source, then it is normally desired to have them deleted when the source is deleted, but also have them deleted when they are no longer referenced from the source. JPA 1.0 did not provide an option for this, so when a dependent object was removed from the source relationship, it had to be explicitly removed from the `EntityManager`. JPA 2.0 provides a `orphanRemoval` option on the `OneToMany`<sup>13</sup> and `OneToOne`<sup>14</sup> annotations and XML. Orphan removal will ensure that any object no longer referenced from the relationship is deleted from the database.

### 30.3.1 Example of an orphan removal one to many relationship annotations

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToMany(orphanRemoval=true, cascade={CascadeType.ALL})
    private List<PhoneNumbers> phones;
    ...
}
```

---

13 <https://java.sun.com/javase/6/docs/api/javax/persistence/OneToMany.html>

14 <https://java.sun.com/javase/6/docs/api/javax/persistence/OneToOne.html>

### 30.3.2 Example of an orphan removal one to many relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <one-to-many name="phones" orphan-removal="true">
      <cascade>
        <cascade-all/>
      </cascade>
    </one-to-many>
  </attributes>
</entity>
```

## 30.4 Target Entity

Relationship mappings have a `targetEntity` attribute that allows the reference class (target) of the relationship to be specified. This is normally not required to be set as it is defaulted from the field type, get method return type, or collection's generic type.

This can also be used if your field uses a public interface type, for example field is interface `Address`, but the mapping needs to be to implementation class `AddressImpl`. Another usage is if your field is a superclass type, but you want to map the relationship to a subclass.

### 30.4.1 Example of a target entity relationship annotations

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToMany(targetEntity=Phone.class)
    @JoinColumn(name="OWNER_ID")
    private List phones;
    ...
}
```

### 30.4.2 Example of a target entity relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <one-to-many name="phones" target-entity="org.acme.Phone">
      <join-column name="OWNER_ID"/>
    </one-to-many>
  </attributes>
</entity>
```

## 30.5 Collections

Collection mappings include `OneToMany`<sup>15</sup>, `ManyToMany`<sup>16</sup>, and in JPA 2.0 `ElementCollection`<sup>17</sup>. JPA requires that the type of the collection field or get/set methods be one of the Java collection interfaces, `Collection`, `List`, `Set`, or `Map`.

### 30.5.1 Collection Implementations

Your field should not be of a collection implementation type, such as `ArrayList`. Some JPA providers may support using collection implementations, many support **EAGER** collection relationships to use the implementation class. You can set any implementation as the instance value of the collection, but when reading an object from the database, if it is **LAZY** the JPA provider will normally put in a special **LAZY** collection.

### 30.5.2 Duplicates

A `List` in Java supports duplicate entries, and a `Set` does not. In the database, duplicates are generally not supported. Technically it could be possible if a `JoinTable` is used, but JPA does not require duplicates to be supported, and most providers do not.

If you require duplicate support, you may need to create an object that represents and maps to the join table. This object would still require a unique `Id`, such as a `GeneratedValue`. See Mapping a Join Table with Additional Columns<sup>18</sup>.

### 30.5.3 Ordering

JPA allows the collection values to be ordered by the database when retrieved. This is done through the `@OrderBy`<sup>19</sup> annotation or `<order-by>` XML element.

The value of the `OrderBy` is a JPQL `ORDER BY`<sup>20</sup> string. The can be an attribute name followed by `ASC` or `DESC` for ascending or descending ordering. You could also use a path or nested attribute, or a `"`, `"` for multiple attributes. If no `OrderBy` value is given it is assumed to be the `Id` of the target object.

The `OrderBy` value must be a mapped attribute of the target object. If you want to have an ordered `List` you need to add an *index* attribute to your target object and an *index* column to it's table. You will also have to ensure you set the index values. JPA 2.0 will have extended support for an ordered `List` using an `OrderColumn`.

Note that using an `OrderBy` does not ensure the collection is ordered in memory. You are responsible for adding to the collection in the correct order. Java does define a `SortedSet`

---

15 Chapter 35.1 on page 169

16 Chapter 37.1.2 on page 178

17 Chapter 39.1.2 on page 185

18 Chapter 39.1 on page 183

19 <https://java.sun.com/javaee/5/docs/api/javax/persistence/OrderBy.html>

20 Chapter 46.1.6 on page 237

interface and `TreeSet` collection implementation that does maintain an order. JPA does not specifically support `SortedSet`, but some JPA providers may allow you to use a `SortedSet` or `TreeSet` for your collection type, and maintain the correct ordering. By default these require your target object to implement the `Comparable` interface, or set a `Comparator`. You can also use the `Collections.sort()` method to sort a `List` when required. One option to sort in memory is to use property access and in your set and add methods call `Collections.sort()`.

### Example of a collection order by annotation

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToMany
    @OrderBy("areaCode")
    private List<Phone> phones;
    ...
}
```

### Example of a collection order by XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
    <attributes>
        <id name="id"/>
        <one-to-many name="phones">
            <order-by>areaCode</order-by>
        </one-to-many>
    </attributes>
</entity>
```

## 30.5.4 Order Column (JPA 2.0)

JPA 2.0 adds support for an `OrderColumn`. An `OrderColumn` can be used to define an order `List` on any collection mapping. It is defined through the `@OrderColumn`<sup>21</sup> annotation or `<order-column>` XML element.

The `OrderColumn` is maintained by the mapping and should not be an attribute of the target object. The table for the `OrderColumn` depends on the mapping. For a `OneToMany` mapping it will be in the target object's table. For a `ManyToMany` mapping or a `OneToMany` using a `JoinTable` it will be in the join table. For an `ElementCollection` mapping it will be in the target table.

### Example of a collection order column database

EMPLOYEE (table)

<sup>21</sup> <https://java.sun.com/javaee/6/docs/api/javax/persistence/OrderColumn.html>

ID	FIRSTNAME	LASTNAME	SALARY
1	Bob	Way	50000
2	Sarah	Smith	60000

EMPLOYEE\_PHONE (table)

EMPLOYEE_ID	PHONE_ID	INDEX
1	1	0
1	3	1
2	2	0
2	4	1

PHONE(table)

ID	AREACODE	NUMBER
1	613	792-7777
2	416	798-6666
3	613	792-9999
4	416	798-5555

### Example of a collection order column annotation

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToMany
    @OrderColumn(name="INDEX")
    private List<Phone> phones;
    ...
}
```

### Example of a collection order column XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <one-to-many name="phones">
      <order-column name="INDEX"/>
    </one-to-many>
  </attributes>
</entity>
```

## 30.6 Common Problems

*Object corruption, one side of the relationship is not updated after updating the other side*

A common problem with bi-directional relationships is the application updates one side of the relationship, but the other side does not get updated, and becomes out of sync. In JPA, as in Java in general it is the responsibility of the application, or the object model to maintain relationships. If your application adds to one side of a relationship, then it must add to the other side.

This is commonly resolved through `add` or `set` methods in the object model that handle both sides of the relationships, so the application code does not need to worry about it. There are two ways to go about this, you can either only add the relationship maintenance code to one side of the relationship, and only use the setter from one side (such as making the other side protected), or add it to both sides and ensure you avoid a infinite loop.

For example:

```
public class Employee {
    private List phones;
    ...
    public void addPhone(Phone phone) {
        this.phones.add(phone);
        if (phone.getOwner() != this) {
            phone.setOwner(this);
        }
    }
    ...
}

public class Phone {
    private Employee owner;
    ...
    public void setOwner(Employee employee) {
        this.owner = employee;
        if (!employee.getPhones().contains(this)) {
            employee.getPhones().add(this);
        }
    }
    ...
}
```

The code is similar for bi-directional `OneToOne` and `ManyToMany` relationships.

Some expect the JPA provider to have magic that automatically maintains relationships. This was actually part of the EJB CMP 2 specification. However the issue is if the objects are detached or serialized to another VM, or new objects are related before being managed, or the object model is used outside the scope of JPA, then the magic is gone, and the application is left figuring things out, so in general it may be better to add the code to the object model. However some JPA providers do have support for automatically maintaining relationships.

In some cases it is undesirable to instantiate a large collection when adding a child object. One solution is to not map the bi-directional relationship, and instead query for it as required.

Also some JPA providers optimize their lazy collection objects to handle this case, so you can still add to the collection without instantiating it.

### *Poor performance, excessive queries*

This most common issue leading to poor performance is the usage of **EAGER** relationships. This requires the related objects to be read when the source objects are read. So for example reading the president of the company with **EAGER managedEmployees** will cause every **Employee** in the company to be read. The solution is to always make all relationships **LAZY**. By default **OneToMany** and **ManyToMany** are **LAZY** but **OneToOne** and **ManyToOne** are not, so make sure you configure them to be. See, [Lazy Fetching](#)<sup>22</sup>. Sometimes you have **LAZY** configured but it does not work, see [Lazy is not working](#)<sup>23</sup>

Another common problems is the  $n+1$  issue. For example consider that you read all **Employee** objects then access their **Address**. Since each **Address** is accessed separately this will cause  $n+1$  queries, which can be a major performance problem. This can be solved through [Join Fetching](#)<sup>24</sup> and [Batch Reading](#)<sup>25</sup>.

### *Lazy is not working*

Lazy **OneToOne** and **ManyToOne** relationships typically require some form of weaving or byte-code generation. Normally when running in JSE an **agent** option is required to allow the byte-code weaving, so ensure you have the agent configured correctly. Some JPA providers perform dynamic subclass generation, so do not require an agent.

### **Example agent**

```
java -javaagent:eclipselink.jar ...
```

Some JPA providers also provide static weaving instead, or in addition to dynamic weaving. For static weaving some preprocessor must be run on your JPA classes.

When running in JEE lazy should normally work, as the class loader hook is required by the EJB specification. However some JEE providers may not support this, so static weaving may be required.

Also ensure that you are not accessing the relationship when you shouldn't be. For example if you use property access, and in your set method access the related lazy value, this will cause it to be loaded. Either remove the set method side-effects, or use field access.

---

<sup>22</sup> Chapter 30.1 on page 126

<sup>23</sup> Chapter 31.6 on page 148

<sup>24</sup> Chapter 31.3 on page 145

<sup>25</sup> Chapter 31.4 on page 147

***Broken relationships after serialization***

If your relationship is marked as `lazy` then if it has not been instantiated before the object is serialized, then it may not get serialized. This may cause an error, or return `null` if it is accessed after deserialization.

See, [Serialization, and Detaching](#)<sup>26</sup>

***Dependent object removed from OneToMany collection is not deleted***

When you remove an object from a collection, if you also want the object deleted from the database you must call `remove()` on the object. In JPA 1.0 even if your relationship is `cascade REMOVE`, you still must call `remove()`, only the remove of the parent object is cascaded, not removal from the collection.

JPA 2.0 will provide an option for having removes from the collection trigger deletion. Some JPA providers support an option for this in JPA 1.0.

See, [Cascading](#)<sup>27</sup>

***My relationship target is an interface***

If your relationship field's type is a public interface of your class, and only has a single implementer, then this is simple to solve, you just need to set a `targetEntity` on your mapping. See, [Target Entity](#)<sup>28</sup>.

If your interface has multiple implementers, then this is more complex. JPA does not directly support mapping interfaces. One solution is to convert the interface to an abstract class and use inheritance to map it. You could also keep the interface, create the abstract class and make sure each implementer extends it, and set the `targetEntity` to be the abstract class.

Another solution is to define virtual attributes using `get/set` methods for each possible implementer, and map these separately, and mark the interface `get/set` as `transient`. You could also not map the attribute, and instead query for it as required.

See, [Variable and Heterogeneous Relationships](#)<sup>29</sup>

Some JPA providers have support for interfaces and variable relationships.

`TopLink`<sup>30</sup>, `EclipseLink`<sup>31</sup> : Support variable relationships through their `@VariableOneToOne` annotation and XML. Mapping to and querying interfaces are also supported through their `ClassDescriptor's InterfacePolicy` API.

---

<sup>26</sup> Chapter 30.1.5 on page 127

<sup>27</sup> Chapter 30.2 on page 129

<sup>28</sup> Chapter 30.4 on page 131

<sup>29</sup> Chapter 31.6 on page 148

<sup>30</sup> Chapter 11 on page 25

<sup>31</sup> Chapter 10 on page 23





# 31 Advanced

## 31.1 Advanced Relationships

### 31.1.1 JPA 2.0 Relationship Enhancements

- `ElementCollection`<sup>1</sup> - A `Collection` or `Map` of `Embeddable` or `Basic` values.
- `Map Columns`<sup>2</sup> - A `OneToMany` or `ManyToMany` or `ElementCollection` that has a `Basic`, `Embeddable` or `Entity` key not part of the target object.
- `Order Columns`<sup>3</sup> - A `OneToMany` or `ManyToMany` or `ElementCollection` can now have a `OrderColumn` that defines the order of the collection when a `List` is used.
- `Unidirectional OneToMany`<sup>4</sup> - A `OneToMany` no longer requires the `ManyToOne` inverse relationship to be defined.

### 31.1.2 Other Types of Relationships

- `Variable`<sup>5</sup> `OneToOne`, `ManyToOne` - A reference to an interface or common unmapped inheritance class that has multiple distinct implementors.
- `Variable`<sup>6</sup> `OneToMany`, `ManyToMany` - A `Collection` or `Map` of heterogeneous objects that share an interface or common unmapped inheritance class that has multiple distinct implementors.
- `Nested collection relationships`<sup>7</sup>, such as an array of arrays, `List` of `Lists`, or `Map` of `Maps`, or other such combinations.
- `Object-Relational Data Type`<sup>8</sup> - Relationships stored in the database using `STRUCT`, `VARRAY`, `REF`, or `NESTEDTABLE` types.
- `XML relationships`<sup>9</sup> - Relationships stored as XML documents.

---

1 Chapter 39.1.2 on page 185

2 Chapter 31.2.1 on page 141

3 Chapter 30.5.4 on page 133

4 Chapter 37.1 on page 177

5 Chapter 31.6 on page 148

6 Chapter 31.6 on page 148

7 Chapter 31.7 on page 149

8 Chapter 41.5 on page 199

9 Chapter 41.6 on page 200

## 31.2 Maps

Java defines the `Map` interface to represent collections whose values are indexed on a key. There are several `Map` implementations, the most common is `HashMap`, but also `Hashtable` and `TreeMap`.

JPA allows a `Map` to be used for any collection mapping including, `OneToMany`, `ManyToMany` and `ElementCollection`. JPA requires that the `Map` interface be used as the attribute type, although some JPA providers may also support using `Map` implementations.

In JPA 1.0 the map key must be a mapped attribute of the collection values. The `@MapKey`<sup>10</sup> annotation or `<map-key>` XML element is used to define a map relationship. If the `MapKey` is not specified it defaults to the target object's `Id`.

### Example of a map key relationship annotation

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToMany(mappedBy="owner")
    @MapKey(name="type")
    private Map<String, PhoneNumber> phoneNumbers;
    ...
}

@Entity
public class PhoneNumber {
    @Id
    private long id;
    @Basic
    private String type; // Either "home", "work", or "fax".
    ...
    @ManyToOne
    private Employee owner;
    ...
}
```

### Example of a map key relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <one-to-many name="phoneNumbers" mapped-by="owner">
      <map-key name="type"/>
    </one-to-many>
  </attributes>
</entity>
<entity name="PhoneNumber" class="org.acme.PhoneNumber"
access="FIELD">
  <attributes>
    <id name="id"/>
    <basic name="type"/>
  </attributes>
</entity>
```

---

10 <https://java.sun.com/javase/5/docs/api/javax/persistence/MapKey.html>

```

    <many-to-one name="owner"/>
  </attributes>
</entity>

```

### 31.2.1 Map Key Columns (JPA 2.0)

JPA 2.0 allows for a `Map` where the key is not part of the target object to be persisted. The `Map` key can be any of the following:

- A `Basic` value, stored in the target's table or join table.
- An `Embedded` object, stored in the target's table or join table.
- A foreign key to another `Entity`, stored in the target's table or join table.

`Map` columns can be used for any collection mapping including, `OneToMany`, `ManyToMany` and `ElementCollection`.

This allows for great flexibility and complexity in the number of different models that can be mapped. The type of mapping used is always determined by the value of the `Map`, not the key. So if the key is a `Basic` but the value is an `Entity` a `OneToMany` mapping is still used. But if the value is a `Basic` but the key is an `Entity` a `ElementCollection` mapping is used.

This allows some very sophisticated database schemas to be mapped. Such as a three way join table, can be mapped using a `ManyToMany` with a `MapKeyJoinColumn` for the third foreign key. For a `ManyToMany` the key is always stored in the `JoinTable`. For a `OneToMany` it is stored in the `JoinTable` if defined, otherwise it is stored in the target `Entity`'s table, even though the target `Entity` does not map this column. For an `ElementCollection` the key is stored in the element's table.

The `@MapKeyColumn`<sup>11</sup> annotation or `<map-key-column>` XML element is used to define a map relationship where the key is a `Basic` value, the `@MapKeyEnumerated`<sup>12</sup> and `@MapKeyTemporal`<sup>13</sup> can also be used with this for `Enum` or `Calendar` types. The `@MapKeyJoinColumn`<sup>14</sup> annotation or `<map-key-join-column>` XML element is used to define a map relationship where the key is an `Entity` value, the `@MapKeyJoinColumns`<sup>15</sup> can also be used with this for composite foreign keys. The annotation `@MapKeyClass`<sup>16</sup> or `<map-key-class>` XML element can be used when the key is an `Embeddable` or to specify the target class or type if generics are not used.

#### Example of a map key column relationship database

EMPLOYEE (table)

ID	FIRSTNAME	LASTNAME	SALARY
----	-----------	----------	--------

- 
- 11 <https://java.sun.com/javaee/6/docs/api/javax/persistence/MapKeyColumn.html>
  - 12 <https://java.sun.com/javaee/6/docs/api/javax/persistence/MapKeyEnumerated.html>
  - 13 <https://java.sun.com/javaee/6/docs/api/javax/persistence/MapKeyTemporal.html>
  - 14 <https://java.sun.com/javaee/6/docs/api/javax/persistence/MapKeyJoinColumn.html>
  - 15 <https://java.sun.com/javaee/6/docs/api/javax/persistence/MapKeyJoinColumns.html>
  - 16 <https://java.sun.com/javaee/6/docs/api/javax/persistence/MapKeyClass.html>

1	Bob	Way	50000
2	Sarah	Smith	60000

PHONE(table)

ID	OWNER_ID	PHONE_TYPE	AREACODE	NUMBER
1	1	home	613	792-7777
2	1	cell	613	798-6666
3	2	home	416	792-9999
4	2	fax	416	798-5555

### Example of a map key column relationship annotation

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToMany(mappedBy="owner")
    @MapKeyColumn(name="PHONE_TYPE")
    private Map<String, Phone> phones;
    ...
}
```

```
@Entity
public class Phone {
    @Id
    private long id;
    ...
    @ManyToOne
    private Employee owner;
    ...
}
```

### Example of a map key column relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <one-to-many name="phones" mapped-by="owner">
      <map-key-column name="PHONE_TYPE"/>
    </one-to-many>
  </attributes>
</entity>
<entity name="Phone" class="org.acme.Phone" access="FIELD">
  <attributes>
    <id name="id"/>
    <many-to-one name="owner"/>
  </attributes>
</entity>
```

### Example of a map key join column relationship database

EMPLOYEE (table)

ID	FIRSTNAME	LASTNAME	SALARY
1	Bob	Way	50000
2	Sarah	Smith	60000

PHONE(table)

ID	OWNER_ID	PHONE_TYPE_ID	AREACODE	NUMBER
1	1	1	613	792-7777
2	1	2	613	798-6666
3	2	1	416	792-9999
4	2	3	416	798-5555

PHONETYPE(table)

ID	TYPE
1	home
2	cell
3	fax
4	work

### Example of a map key join column relationship annotation

```

@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToMany(mappedBy="owner")
    @MapKeyJoinColumn(name="PHONE_TYPE_ID")
    private Map<PhoneType, Phone> phones;
    ...
}

@Entity
public class Phone {
    @Id
    private long id;
    ...
    @ManyToOne
    private Employee owner;
    ...
}

@Entity
public class PhoneType {
    @Id
    private long id;
    ...
    @Basic
    private String type;
    ...
}

```

### Example of a map key join column relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <one-to-many name="phones" mapped-by="owner">
      <map-key-join-column name="PHONE_TYPE_ID"/>
    </one-to-many>
  </attributes>
</entity>
<entity name="Phone" class="org.acme.Phone" access="FIELD">
  <attributes>
    <id name="id"/>
    <many-to-one name="owner"/>
  </attributes>
</entity>
<entity name="PhoneType" class="org.acme.PhoneType" access="FIELD">
  <attributes>
    <id name="id"/>
    <basic name="type"/>
  </attributes>
</entity>
```

### Example of a map key class embedded relationship database

EMPLOYEE (table)

ID	FIRSTNAME	LASTNAME	SALARY
1	Bob	Way	50000
2	Sarah	Smith	60000

EMPLOYEE\_PHONE (table)

EMPLOYEE_ID	PHONE_ID	TYPE
1	1	home
1	2	cell
2	3	home
2	4	fax

PHONE (table)

ID	AREACODE	NUMBER
1	613	792-7777
2	613	798-6666
3	416	792-9999
4	416	798-5555

### Example of a map key class embedded relationship annotation

```
@Entity
public class Employee {
```

```

@Id
private long id;
...
@OneToMany
@MapKeyClass(PhoneType.class)
private Map<PhoneType, Phone> phones;
...
}

@Entity
public class Phone {
    @Id
    private long id;
    ...
}

@Embeddable
public class PhoneType {
    @Basic
    private String type;
    ...
}

```

### Example of a map key class embedded relationship XML

```

<entity name="Employee" class="org.acme.Employee" access="FIELD">
    <attributes>
        <id name="id"/>
        <one-to-many name="phones">
            <map-key-class>PhoneType</map-key-class>
        </one-to-many>
    </attributes>
</entity>
<entity name="Phone" class="org.acme.Phone" access="FIELD">
    <attributes>
        <id name="id"/>
        <many-to-one name="owner"/>
    </attributes>
</entity>
<embeddable name="PhoneType" class="org.acme.PhoneType"
    access="FIELD">
    <attributes>
        <basic name="type"/>
    </attributes>
</embeddable>

```

## 31.3 Join Fetching

Join fetching is a query optimization technique for reading multiple objects in a single database query. It involves joining the two object's tables in SQL and selecting both object's data. Join fetching is commonly used for **OneToOne** relationships, but also can be used for any relationship including **OneToMany** and **ManyToMany**.

Join fetching is one solution to the classic ORM  $n+1$  performance problem. The issue is if you select  $n$  **Employee** objects, and access each of their addresses, in basic ORM (including JPA) you will get  $1$  database select for the **Employee** objects, and then  $n$  database selects, one for each **Address** object. Join fetching solves this issue by only requiring one select, and selecting both the **Employee** and its **Address**.



JPA supports join fetching through JPQL using the `JOIN FETCH` syntax.

### Example of JPQL Join Fetch

```
SELECT emp FROM Employee emp JOIN FETCH emp.address
```

This causes both the `Employee` and `Address` data to be selected in a single query.

#### 31.3.1 Outer Joins

Using the JPQL `JOIN FETCH` syntax a normal `INNER` join is performed. This has the side effect of filtering any `Employee` from the result set that did not have an address. An `OUTER` join in SQL is one that does not filter absent rows on the join, but instead joins a row of all `null` values. If your relationship allows `null` or an empty collection for collection relationships, then you can use an `OUTER` join fetch, this is done in JPQL using the `LEFT` syntax.

Note that `OUTER` joins can be less efficient on some databases, so avoid using an `OUTER` if it is not required.

### Example of JPQL Outer Join Fetch

```
SELECT emp FROM Employee emp LEFT JOIN FETCH emp.address
```

#### 31.3.2 Mapping Level Join Fetch and `EAGER`

JPA has no way to specify that a join fetch always be used for a relationship. Normally it is better to specify the join fetch at the query level, as some use cases may require the related objects, and other use cases may not. JPA does support an `EAGER` option on mappings, but this means that the relationship will be loaded, not that it will be joined. It may be desirable to mark all relationships as `EAGER` as everything is desired to be loaded, but join fetching everything in one huge select could result in a inefficient, overly complex, or invalid join on the database.

Some JPA providers do interpret `EAGER` as join fetch, so this may work on some JPA providers. Some JPA providers support a separate option for always join fetching a relationship.

TopLink<sup>17</sup>, EclipseLink<sup>18</sup> : Support a `@JoinFetch` annotation and XML on a mapping to define that the relationship always be join fetched.

---

<sup>17</sup> Chapter 11 on page 25

<sup>18</sup> Chapter 10 on page 23

### 31.3.3 Nested Joins

JPA 1.0 does not allow nested join fetches in JPQL, although this may be supported by some JPA providers. You can join fetch multiple relationships, but not nested relationships.

#### Example of Multiple JPQL Join Fetch

```
SELECT emp FROM Employee emp LEFT JOIN FETCH emp.address LEFT JOIN
    FETCH emp.phoneNumbers
```

### 31.3.4 Duplicate Data and Huge Joins

One issue with join fetching is that duplicate data can be returned. For example consider join fetching an **Employee**'s **phoneNumbers** relationship. If each **Employee** has 3 **Phone** objects in its **phoneNumbers** collection, the join will require to bring back  $n*3$  rows. As there are 3 phone rows for each employee row, the employee row will be duplicated 3 times. So you are reading more data than if you have selected the objects in  $n+1$  queries. Normally the fact that your executing fewer queries makes up for the fact that you may be reading duplicate data, but if you consider joining multiple collection relationships you can start to get back  $j*i$  duplicate data which can start to become an issue. Even with **ManyToOne** relationships you can be selecting duplicate data. Consider join fetching an **Employee**'s manager, if all or most employee's have the same manager, you will end up select this manager's data many times, in this case you would be better off not using join fetch, and allowing a single query for the manager.

If you start join fetching every relationship, you can start to get some pretty huge joins. This can sometimes be an issue for the database, especially with huge outer joins.

One alternative solution to join fetch that does not suffer from duplicate data is using Batch Fetching<sup>19</sup>.

## 31.4 Batch Fetching

Batch fetching is a query optimization technique for reading multiple related objects in a finite set of database queries. It involves executing the query for the root objects as normal. But for the related objects the original query is joined with the query for the related objects, allowing all of the related objects to be read in a single database query. Batch fetching can be used for any type of relationship.

Batch fetching is one solution to the classic ORM  $n+1$  performance problem. The issue is if you select  $n$  **Employee** objects, and access each of their addresses, in basic ORM (including JPA) you will get 1 database select for the **Employee** objects, and then  $n$  database selects, one for each **Address** object. Batch fetching solves this issue by requiring one select for the **Employee** objects and one select for the **Address** objects.

---

<sup>19</sup> Chapter 31.4 on page 147

Batch fetching is more optimal for reading collection relationships and multiple relationships as it does not require selecting duplicate data as in join fetching.

JPA does not support batch reading, but some JPA providers do.

TopLink<sup>20</sup>, EclipseLink<sup>21</sup> : Support a `@BatchFetch` annotation and xml element and a "eclipselink.batch" query hint to enable batch reading. Three forms of batch fetching are supported, JOIN, EXISTS, and IN.

See also,

- Batch fetching - optimizing object graph loading <sup>22</sup>

## 31.5 Filtering, Complex Joins

Normally a relationship is based on a foreign key in the database, but on occasion it is always based on other conditions. Such as `Employee` having many `PhoneNumbers` but also a single home phone, or one of his phones that has the "home" type, or a collection of "active" projects, or other such condition.

JPA does not support mapping these types of relationships, as it only supports mappings defined by foreign keys, not based on other columns, constant values, or functions. Some JPA providers may support this. Workarounds include, mapping the foreign key part of the relationship, then filtering the results in your get/set methods of your object. You could also query for the results, instead of defining a relationship.

TopLink<sup>23</sup>, EclipseLink<sup>24</sup> : Support filtering and complex relationships through several mechanisms. You can use a `DescriptorCustomizer` to define a `selectionCriteria` on any mapping using the `Expression` criteria API. This allows for any condition to be applied including constants, functions, or complex joins. You can also use a `DescriptorCustomizer` to define the SQL or define a `StoredProcedureCall` for the mapping's `selectionQuery`.

## 31.6 Variable and Heterogeneous Relationships

It is sometimes desirable to define a relationship where the type of the relationship can be one of several unrelated, heterogeneous values. This could either be a `OneToOne`, `ManyToOne`, `OneToMany` or `ManyToMany` relationship. The related values may share a common interface, or may share nothing in common other than subclassing `Object`. It is also possible to conceive of a relationship that could also be any `Basic` value, or even an `Embeddable`.

---

<sup>20</sup> Chapter 11 on page 25

<sup>21</sup> Chapter 10 on page 23

<sup>22</sup> <http://java-persistence-performance.blogspot.com/2010/08/batch-fetching-optimizing-object-graph.html>

<sup>23</sup> Chapter 11 on page 25

<sup>24</sup> Chapter 10 on page 23

In general JPA does not support variable, interface, or heterogeneous relationships. JPA does support relationships to inheritance classes, so the easiest workaround is normally to define a common superclass for the related values.

Another solution is to define virtual attributes using get/set methods for each possible implementer, and map these separately, and mark the heterogeneous get/set as **transient**. You could also not map the attribute, and instead query for it as required.

For heterogeneous **Basic** or **Embeddable** relationships one solution is to serialize the value to a binary field. You could also convert the value to a **String** representation that you can restore the value from, or store the value to two columns, one storing the **String** value and the other the class name or type.

Some JPA providers have support for interfaces, variable relationships, and/or heterogeneous relationships.

TopLink<sup>25</sup>, EclipseLink<sup>26</sup> : Support variable relationships through their **@VariableOneToOne** annotation and XML. Mapping to and querying interfaces are also supported through their **ClassDescriptor**'s **InterfacePolicy** API.

## 31.7 Nested Collections, Maps and Matrices

It is somewhat common in an object model to have complex collection relationships such as a **List of Lists** (i.e. a matrix), or a **Map of Maps**, or a **Map of Lists**, and so on. Unfortunately these types of collections map very poorly to a relational database.

JPA does not support nested collection relationships, and normally it is best to change your object model to avoid them to make persistence and querying easier. One solution is to create an object that wraps the nested collection.

For example if an **Employee** had a **Map of Projects** keyed by a **String** project-type and the value a **List of Projects**. To map this a new **ProjectType** class could be created to store the project-type and a **OneToMany** to **Project**.

### Example nested collection model (original)

```
public class Employee {  
    private long id;  
    private Map<String, List<Project>> projects;  
}
```

### Example nested collection model (modified)

```
public class Employee {  
    @Id  
    @GeneratedValue
```

---

25 Chapter 11 on page 25

26 Chapter 10 on page 23

```
private long id;
...
@OneToMany(mappedBy="employee")
@MapKey(name="type")
private Map<String, ProjectType> projects;
}

public class ProjectType {
    @Id
    @GeneratedValue
    private long id;
    @ManyToOne
    private Employee employee;
    @Column(name="PROJ_TYPE")
    private String type;
    @ManyToMany
    private List<Project> projects;
}
```

### Example nested collection database

EMPLOYEE (table)

ID	FIRSTNAME	LASTNAME	SALARY
1	Bob	Way	50000
2	Sarah	Smith	60000

PROJECTTYPE (table)

ID	EMPLOYEE_ID	PROJ__TYPE
1	1	small
2	1	medium
3	2	large

PROJECTTYPE\_PROJECT (table)

PROJECTTYPE_ID	PROJECT_ID
1	1
1	2
2	3
3	4

Relationships<sup>27</sup> Relationships<sup>28</sup>

---

<sup>27</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

<sup>28</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FMapping>

## 32 OneToOne

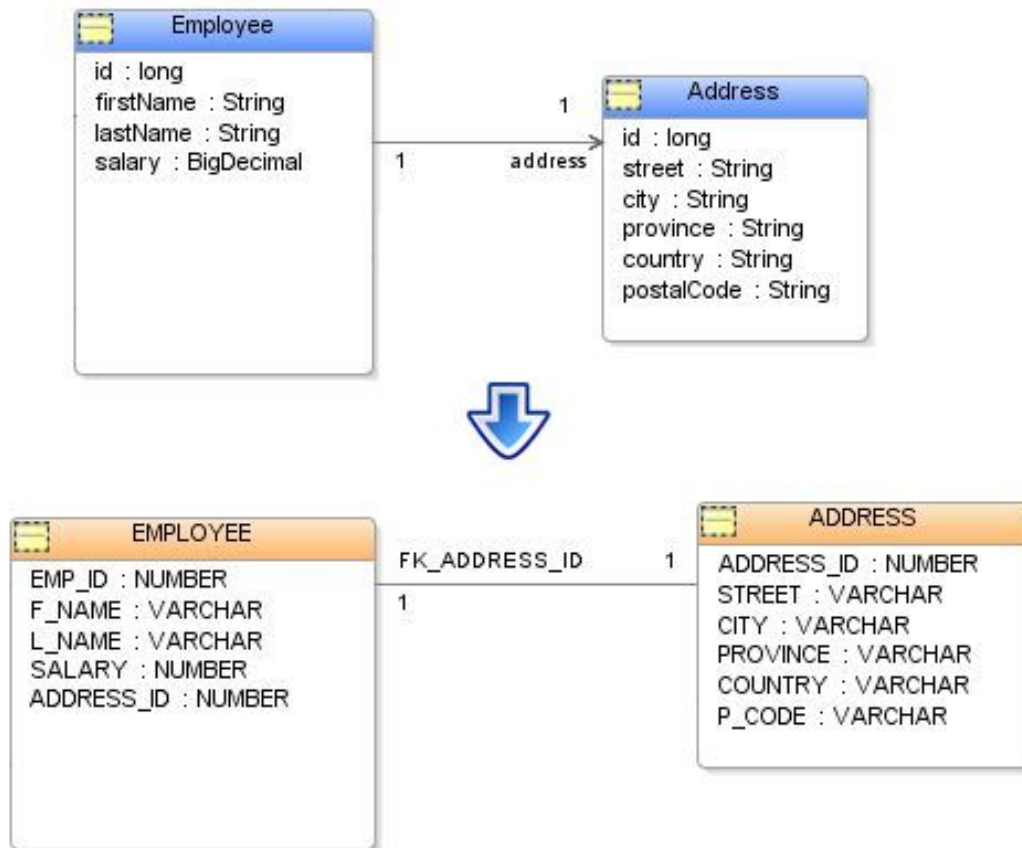


Figure 14

A **OneToOne** relationship in Java is where the source object has an attribute that references another target object and (if) that target object had the inverse relationship back to the source object it would also be a **OneToOne** relationship. All relationships in Java and JPA are unidirectional, in that if a source object references a target object there is no guarantee that the target object also has a relationship to the source object. This is different than a relational database, in which relationships are defined through foreign keys and querying such that the inverse query always exists.

JPA also defines a **ManyToOne**<sup>1</sup> relationship, which is similar to a **OneToOne** relationship except that the inverse relationship (if it were defined) is a **OneToMany** relationship. The main

<sup>1</sup> Chapter 33.2.2 on page 164

difference between a **OneToOne** and a **ManyToOne** relationship in JPA is that a **ManyToOne** always contains a foreign key from the source object's table to the target object's table, where as a **OneToOne** relationship the foreign key may either be in the source object's table or the target object's table. If the foreign key is in the target object's table JPA requires that the relationship be bi-directional (must be defined in both objects), and the source object must use the `mappedBy` attribute to define the mapping.

In JPA a **OneToOne** relationship is defined through the `@OneToOne`<sup>2</sup> annotation or the `<one-to-one>` element. A **OneToOne** relationship typically requires a `@JoinColumn`<sup>3</sup> or `@JoinColumns`<sup>4</sup> if a composite primary key.

### Example of a OneToOne relationship database

EMPLOYEE (table)

EMP_ID	FIRSTNAME	LASTNAME	SALARY	ADDRESS_ID
1	Bob	Way	50000	6
2	Sarah	Smith	60000	7

ADDRESS (table)

---

2 <https://java.sun.com/javaee/5/docs/api/javax/persistence/OneToOne.html>  
3 <https://java.sun.com/javaee/5/docs/api/javax/persistence/JoinColumn.html>  
4 <https://java.sun.com/javaee/5/docs/api/javax/persistence/JoinColumns.html>

ADDRESS_ID	STREET	CITY	PROVINCE	COUNTRY	P_CODE
6	17 Bank St	Ottawa	ON	Canada	K2H7Z5
7	22 Main St	Toronto	ON	Canada	L5H2D5



### 32.0.1 Example of a OneToOne relationship annotations

```
@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private long id;
    ...
    @OneToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="ADDRESS_ID")
    private Address address;
    ...
}
```

### 32.0.2 Example of a OneToOne relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id">
      <column name="EMP_ID"/>
    </id>
    <one-to-one name="address" fetch="LAZY">
      <join-column name="ADDRESS_ID"/>
    </one-to-one>
  </attributes>
</entity>
```

## 32.1 Inverse Relationships, Target Foreign Keys and Mapped By

A typical object centric perspective of a **OneToOne** relationship has the data model mirror the object model, in that the *source* object has a pointer to the *target* object, so the database *source* table has a foreign key to the *target* table. This is not how things always work out in the database though, in fact many database developers would think having the foreign key in the *target* table to be logical, as this enforces the uniqueness of the **OneToOne** relationship. Personally I prefer the object perspective, however you will most likely encounter both.

To start with consider a bi-directional **OneToOne** relationship, you do not require two foreign keys, one in each table, so a single foreign key in the *owning* side of the relationship is sufficient. In JPA the *inverse OneToOne must* use the **mappedBy** attribute (with some exceptions<sup>5</sup>), this makes the JPA provider use the foreign key and mapping information in the *source* mapping to define the *target* mapping.

See also, Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys<sup>6</sup>.

The following gives an example of what the inverse **address** relationship would look like.

---

<sup>5</sup> Chapter 33.1 on page 157

<sup>6</sup> Chapter 33.1 on page 157

### 32.1.1 Example of an inverse OneToOne relationship annotations

```
@Entity
public class Address {
    @Id
    private long id;
    ...
    @OneToOne(fetch=FetchType.LAZY, mappedBy="address")
    private Employee owner;
    ...
}
```

### 32.1.2 Example of an inverse OneToOne relationship XML

```
<entity name="Address" class="org.acme.Address" access="FIELD">
    <attributes>
        <id name="id"/>
        <one-to-one name="owner" fetch="LAZY" mapped-by="address"/>
    </attributes>
</entity>
```

## 32.2 See Also

- Relationships<sup>7</sup>
  - Cascading<sup>8</sup>
  - Lazy Fetching<sup>9</sup>
  - Target Entity<sup>10</sup>
  - Join Fetching<sup>11</sup>
  - Batch Reading<sup>12</sup>
  - Common Problems<sup>13</sup>
- ManyToOne<sup>14</sup>

## 32.3 Common Problems

*Foreign key is also part of the primary key.*

See Primary Keys through OneToOne Relationships<sup>15</sup>.

---

7 Chapter 29.9 on page 123  
 8 Chapter 30.2 on page 129  
 9 Chapter 30.1 on page 126  
 10 Chapter 30.4 on page 131  
 11 Chapter 31.3 on page 145  
 12 Chapter 31.6 on page 148  
 13 Chapter 30.6 on page 135  
 14 Chapter 33.2.2 on page 164  
 15 Chapter 23.2 on page 69

***Foreign key is also mapped as a basic.***

If you use the same field in two different mappings, you typically require to make one of them read-only using `insertable, updatable = false`.

See Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys<sup>16</sup>.

***Constraint error on insert.***

This typically occurs because you have incorrectly mapped the foreign key in a `OneToOne` relationship.

See Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys<sup>17</sup>.

It can also occur if your JPA provider does not support referential integrity, or does not resolve bi-directional constraints. In this case you may either need to remove the constraint, or use `EntityManager.flush()` to ensure the order your objects are written in.

***Foreign key value is null***

Ensure you set the value of the object's `OneToOne`, if the `OneToOne` is part of a bi-directional `OneToOne` relationship, ensure you set `OneToOne` in both object's, JPA does not maintain bi-directional relationships for you.

Also check that you defined the `JoinColumn` correctly, ensure you did not set `insertable, updatable = false` or use a `PrimaryKeyJoinColumn`, or `mappedBy`.

---

<sup>16</sup> Chapter 33.1 on page 157

<sup>17</sup> Chapter 33.1 on page 157

## 33 Advanced

### 33.1 Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys

If a **OneToOne** relationship uses a *target foreign key* (the foreign key is in the target table, not the source table), then JPA requires that you define a **OneToOne** mapping in both directions, and that the *target foreign key* mapping use the **mappedBy** attribute. The reason for this, is the mapping in the source object only affects the row the JPA writes to the source table, if the foreign key is in the target table, JPA has no easy way to write this field.

There are other ways around this problem however. In JPA the **JoinColumn** defines an **insertable** and **updatable** attribute, these can be used to instruct the JPA provider that the foreign key is actually in the target object's table. With these enabled JPA will not write anything to the source table, most JPA providers will also infer that the foreign key constraint is in the target table to preserve referential integrity on insertion. JPA also defines the **@PrimaryKeyJoinColumn**<sup>1</sup> that can be used to define the same thing. You still must map the foreign key in the target object in some fashion though, but could just use a **Basic** mapping to do this.

Some JPA providers may support an option for a unidirectional **OneToOne** mapping for target foreign keys.

Target foreign keys can be tricky to understand, so you might want to read this section twice. They can get even more complex though. If you have a data model that cascades primary keys then you can end up with a single **OneToOne** that has both a logical foreign key, but has some fields in it that are logically target foreign keys.

For example consider **Company**, **Department**, **Employee**. **Company**'s id is **COM\_ID**, **Department**'s id is a composite primary key of **COM\_ID** and **DEPT\_ID**, and **Employee**'s id is a composite primary key of **COM\_ID**, **DEP\_ID**, and **EMP\_ID**. So for an **Employee** its relationship to **company** uses a normal **ManyToOne** with a foreign key, but its relationship to **department** uses a **ManyToOne** with a foreign key, but the **COM\_ID** uses **insertable = false** or **PrimaryKeyJoinColumn**, because it is actually mapped through the **company** relationship. The **Employee**'s relationship to its **address** then uses a normal foreign key for **ADD\_ID** but a target foreign key for **COM\_ID**, **DEP\_ID**, and **EMP\_ID**.

This may work in some JPA providers, others may require different configuration, or not support this type of data model.

---

<sup>1</sup> <https://java.sun.com/javase/5/docs/api/javax/persistence/PrimaryKeyJoinColumn.html>

### Example of cascaded primary keys database

COMPANY(table)

COM_ID	NAME
1	ACME
2	Wikimedia

DEPARTMENT(table)

COM_ID	DEP_ID	NAME
1	1	Billing
1	2	Research
2	1	Accounting
2	2	Research

EMPLOYEE (table)

COM_ID	DEP_ID	EMP_ID	NAME	MNG_ID	ADD_ID
1	1	1	Bob Way	null	1
1	1	2	Joe Smith	1	2
1	2	1	Sarah Way	null	1
1	2	2	John Doe	1	2
2	1	1	Jane Doe	null	1
2	2	1	Alice Smith	null	1

ADDRESS(table)

COM_ID	DEP_ID	ADD_ID	ADDRESS
1	1	1	17 Bank, Ottawa, ONT
1	1	2	22 Main, Ottawa, ONT
1	2	1	255 Main, Toronto, ONT
1	2	2	12 Main, Winnipeg, MAN
2	1	1	72 Riverside, Winnipeg, MAN
2	2	1	82 Riverside, Winnipeg, MAN

### 33.1.1 Example of cascaded primary keys and mixed OneToOne and ManyToOne mapping annotations

```
@Entity
@IdClass(EmployeeId.class)
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private long employeeId;
    @Id
    @Column(name="DEP_ID", insertable=false, updatable=false)
    private long departmentId;
    @Id
    @Column(name="COM_ID", insertable=false, updatable=false)
    private long companyId;
    ...
    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="COM_ID")
    private Company company;
    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumns({
        @JoinColumn(name="DEP_ID"),
        @JoinColumn(name="COM_ID", insertable=false, updatable=false)
    })
    private Department department;
    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumns({
        @JoinColumn(name="MNG_ID"),
        @JoinColumn(name="DEP_ID", insertable=false, updatable=false),
        @JoinColumn(name="COM_ID", insertable=false, updatable=false)
    })
    private Employee manager;
    @OneToOne(fetch=FetchType.LAZY)
    @JoinColumns({
        @JoinColumn(name="ADD_ID")
        @JoinColumn(name="DEP_ID", insertable=false, updatable=false),
        @JoinColumn(name="COM_ID", insertable=false, updatable=false)
    })
    private Address address;
    ...
}
```

### 33.1.2 Example of cascaded primary keys and mixed OneToOne and ManyToOne mapping annotations using PrimaryKeyJoinColumn

```
@Entity
@IdClass(EmployeeId.class)
```

```

public class Employee {
    @Id
    @Column(name="EMP_ID")
    private long employeeId;
    @Id
    @Column(name="DEP_ID", insertable=false, updatable=false)
    private long departmentId;
    @Id
    @Column(name="COM_ID", insertable=false, updatable=false)
    private long companyId;
    ...
    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="COM_ID")
    private Company company;
    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="DEP_ID")
    @PrimaryKeyJoinColumn(name="COM_ID")
    private Department department;
    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="MNG_ID")
    @PrimaryKeyJoinColumns({
        @PrimaryKeyJoinColumn(name="DEP_ID")
        @PrimaryKeyJoinColumn(name="COM_ID")
    })
    private Employee manager;
    @OneToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="ADD_ID")
    @PrimaryKeyJoinColumns({
        @PrimaryKeyJoinColumn(name="DEP_ID")
        @PrimaryKeyJoinColumn(name="COM_ID")
    })
    private Address address;
    ...
}

```

## 33.2 Mapping a OneToOne Using a Join Table

In some data models, you may have a **OneToOne** relationship defined through a *join table*. For example consider you had existing **EMPLOYEE** and **ADDRESS** tables with no foreign key, and wanted to define a **OneToOne** relationship without changing the existing tables. To do this you could define an intermediate table that contained the primary key of both objects. This is similar to a **ManyToMany** relationship, but if you add a unique constraint to each foreign key you can enforce that it is **OneToOne** (or even **OneToMany**).

JPA defines a join table using the `@JoinTable`<sup>2</sup> annotation and `<join-table>` XML element. A `JoinTable` can be used on a **ManyToMany** or **OneToMany** mappings, but the JPA 1.0 specification is vague whether it can be used on a **OneToOne**. The `JoinTable` documentation does not state that it can be used in a **OneToOne**, but the XML schema for `<one-to-one>` does allow a nested `<join-table>` element. Some JPA providers may support this, and others may not.

If your JPA provider does not support this, you can workaround the issue by instead defining a **OneToMany** or **ManyToMany** relationship and just define a `get/set` method that returns/sets the first element on the collection.

2 <https://java.sun.com/javaee/5/docs/api/javax/persistence/JoinTable.html>



### Example of a OneToOne using a JoinTable database

EMPLOYEE (table)

EMP_ID	FIRSTNAME	LASTNAME	SALARY
1	Bob	Way	50000
2	Sarah	Smith	60000

EMP\_ADD(table)

EMP_ID	ADDR_ID
1	6
2	7

ADDRESS (table)

ADDRESS_ID	STREET	CITY	PROVINCE	COUNTRY	P_CODE
6	17 Bank St	Ottawa	ON	Canada	K2H7Z5
7	22 Main St	Toronto	ON	Canada	L5H2D5

### 33.2.1 Example of a OneToOne using a JoinTable

```
@OneToOne(fetch=FetchType.LAZY)
@JoinTable(
    name="EMP_ADD"
    joinColumns=
        @JoinColumn(name="EMP_ID", referencedColumnName="EMP_ID"),
    inverseJoinColumns=
        @JoinColumn(name="ADDR_ID",
referencedColumnName="ADDRESS_ID"))
private Address address;
...
```

### 33.2.2 Example of simulating a OneToOne using a OneToMany JoinTable

```
@OneToMany
@JoinTable(
    name="EMP_ADD"
    joinColumns=
        @JoinColumn(name="EMP_ID", referencedColumnName="EMP_ID"),
    inverseJoinColumns=
        @JoinColumn(name="ADDR_ID",
referencedColumnName="ADDRESS_ID"))
private List<Address> addresses;
...
public Address getAddress() {
    if (this.addresses.isEmpty()) {
        return null;
    }
    return this.addresses.get(0);
}
public void setAddress(Address address) {
    if (this.addresses.isEmpty()) {
        this.addresses.add(address);
    } else {
        this.addresses.set(1, address);
    }
}
...
```

OneToOne<sup>3</sup> OneToOne<sup>4</sup>

---

<sup>3</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

<sup>4</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FMapping>

## 34 ManyToOne

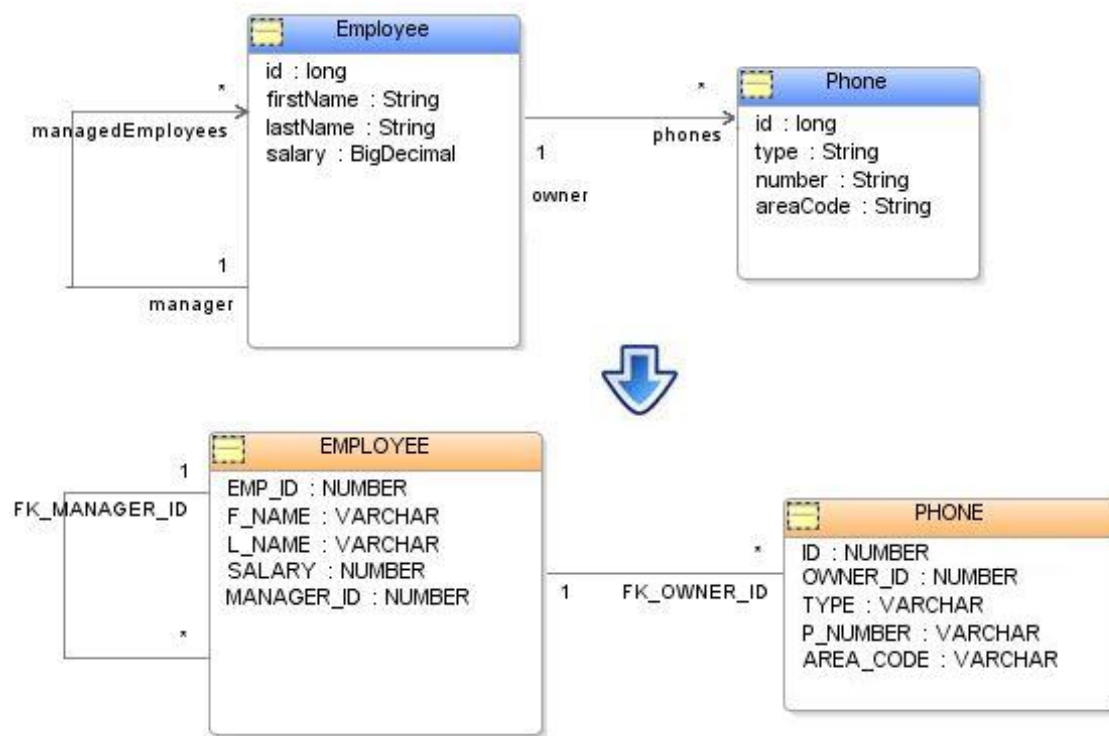


Figure 15

A **ManyToOne** relationship in Java is where the source object has an attribute that references another target object and (if that target object had the inverse relationship back to the source object it would be a **OneToMany** relationship. All relationships in Java and JPA are unidirectional, in that if a source object references a target object there is no guarantee that the target object also has a relationship to the source object. This is different than a relational database, in which relationships are defined through foreign keys and querying such that the inverse query always exists.

JPA also defines a **OneToOne**<sup>1</sup> relationship, which is similar to a **ManyToOne** relationship except that the inverse relationship (if it were defined) is a **OneToOne** relationship. The main difference between a **OneToOne** and a **ManyToOne** relationship in JPA is that a **ManyToOne** always contains a foreign key from the source object's table to the target object's table, whereas a **OneToOne** relationship the foreign key may either be in the source object's table or the target object's table.

<sup>1</sup> Chapter 31.7 on page 150

In JPA a `ManyToOne` relationship is defined through the `@ManyToOne2` annotation or the `<many-to-one>` element.

In JPA a `ManyToOne` relationship is always (well almost always) required to define a `OneToMany3` relationship, the `ManyToOne` always defines the foreign key (`JoinColumn`) and the `OneToMany` must use a `mappedBy` to define its inverse `ManyToOne`.

### Example of a `ManyToOne` relationship database

EMPLOYEE (table)

EMP_ID	FIRSTNAME	LASTNAME	SALARY	MANAGER_ID
1	Bob	Way	50000	2
2	Sarah	Smith	75000	null

PHONE (table)

ID	TYPE	AREA_CODE	P_NUMBER	OWNER_ID
1	home	613	792-0000	1
2	work	613	896-1234	1
3	work	416	123-4444	2

#### 34.0.3 Example of a `ManyToOne` relationship annotations

```
@Entity
public class Phone {
    @Id
    private long id;
    ...
    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="OWNER_ID")
    private Employee owner;
    ...
}
```

#### 34.0.4 Example of a `ManyToOne` relationship XML

```
<entity name="Phone" class="org.acme.Phone" access="FIELD">
    <attributes>
        <id name="id"/>
        <many-to-one name="owner" fetch="LAZY">
            <join-column name="OWNER_ID"/>
        </many-to-one>
    </attributes>
</entity>
```

---

2 <https://java.sun.com/javase/5/docs/api/javax/persistence/ManyToOne.html>  
 3 Chapter 35.1 on page 169

## 34.1 See Also

- Relationships<sup>4</sup>
  - Cascading<sup>5</sup>
  - Lazy Fetching<sup>6</sup>
  - Target Entity<sup>7</sup>
  - Join Fetching<sup>8</sup>
  - Batch Reading<sup>9</sup>
  - Common Problems<sup>10</sup>
- OneToOne<sup>11</sup>
  - Mapping a OneToOne Using a Join Table<sup>12</sup>
- OneToMany<sup>13</sup>

## 34.2 Common Problems

*Foreign key is also part of the primary key.*

See Primary Keys through OneToOne Relationships<sup>14</sup>.

*Foreign key is also mapped as a basic.*

If you use the same field in two different mappings, you typically require to make one of them read-only using `insertable, updateable = false`.

See Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys<sup>15</sup>.

*Constraint error on insert.*

This typically occurs because you have incorrectly mapped the foreign key in a `OneToOne` relationship.

See Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys<sup>16</sup>.

---

4 Chapter 29.9 on page 123  
5 Chapter 30.2 on page 129  
6 Chapter 30.1 on page 126  
7 Chapter 30.4 on page 131  
8 Chapter 31.3 on page 145  
9 Chapter 31.6 on page 148  
10 Chapter 30.6 on page 135  
11 Chapter 31.7 on page 150  
12 Chapter 33.2 on page 161  
13 Chapter 35.1 on page 169  
14 Chapter 22.1.1 on page 58  
15 Chapter 35.1 on page 169  
16 Chapter 35.1 on page 169

It can also occur if your JPA provider does not support referential integrity, or does not resolve bi-directional constraints. In this case you may either need to remove the constraint, or use `EntityManager flush()` to ensure the order your objects are written in.

### *Foreign key value is null*

Ensure you set the value of the object's `OneToOne`, if the `OneToOne` is part of a bi-directional `OneToMany` relationship, ensure you set the object's `OneToOne` when adding an object to the `OneToMany`, JPA does not maintain bi-directional relationships for you.

Also check that you defined the `JoinColumn` correctly, ensure you did not set `insertable`, `updateable = false` or use a `PrimaryKeyJoinColumn`.

## 35 Advanced

### 35.1 Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys

In complex data models it may be required to use a target foreign key, or read-only `JoinColumn` in mapping a `ManyToOne` if the foreign key/`JoinColumn` is shared with other `ManyToOne` or `Basic` mappings.

See, Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys<sup>1</sup>

`ManyToOne`<sup>2</sup> `ManyToOne`<sup>3</sup>

---

<sup>1</sup> Chapter 33.1 on page 157

<sup>2</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

<sup>3</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FMapping>





## 36 OneToMany

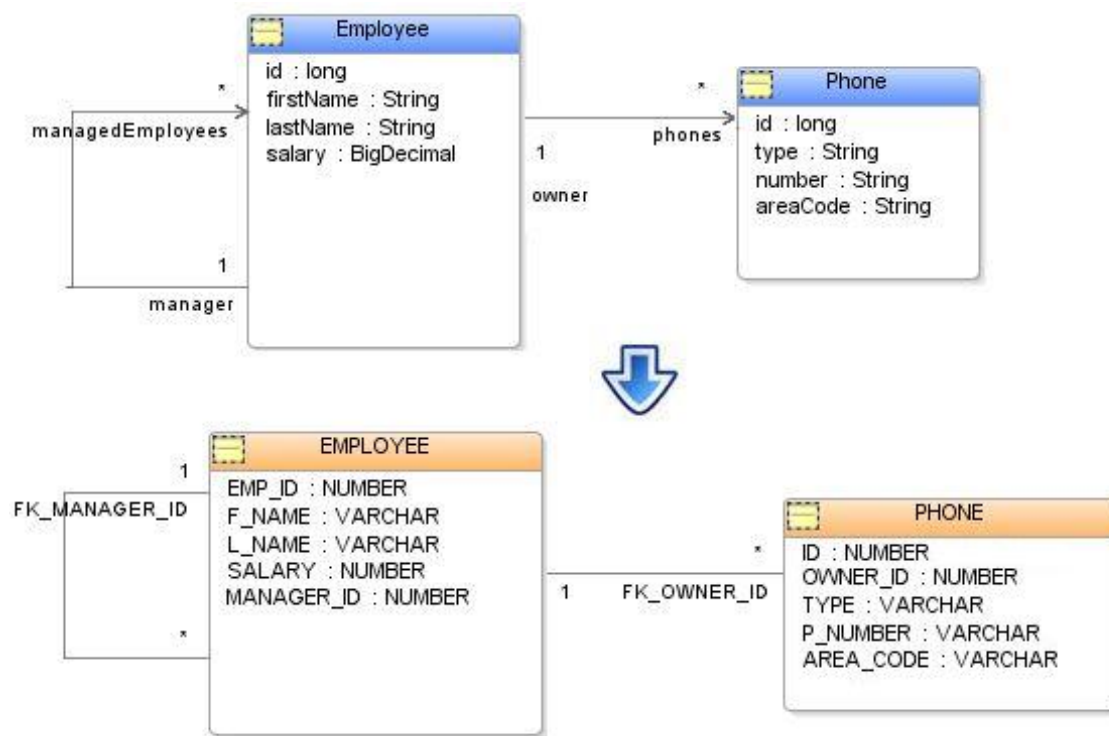


Figure 16

A **OneToMany** relationship in Java is where the source object has an attribute that stores a collection of target objects and (if) those target objects had the inverse relationship back to the source object it would be a **ManyToOne** relationship. All relationships in Java and JPA are unidirectional, in that if a source object references a target object there is no guarantee that the target object also has a relationship to the source object. This is different than a relational database, in which relationships are defined through foreign keys and querying such that the inverse query always exists.

JPA also defines a **ManyToMany**<sup>1</sup> relationship, which is similar to a **OneToMany** relationship except that the inverse relationship (if it were defined) is a **ManyToMany** relationship. The main difference between a **OneToMany** and a **ManyToMany** relationship in JPA is that a **ManyToMany** always makes use of an intermediate relational join table to store the relationship, whereas a **OneToMany** can either use a join table, or a foreign key in target object's table referencing the source object table's primary key. If the **OneToMany** uses a foreign key in the

<sup>1</sup> Chapter 37.1.2 on page 178

target object's table JPA requires that the relationship be bi-directional (inverse **ManyToOne** relationship must be defined in the target object), and the source object must use the **mappedBy** attribute to define the mapping.

In JPA a **OneToMany** relationship is defined through the **@OneToMany**<sup>2</sup> annotation or the **<one-to-many>** element.

### 36.0.1 Example of a OneToMany relationship database

EMPLOYEE (table)

EMP_ID	FIRSTNAME	LASTNAME	SALARY	MANAGER_ID
1	Bob	Way	50000	2
2	Sarah	Smith	75000	null

PHONE (table)

ID	TYPE	AREA_CODE	P_NUMBER	OWNER_ID
1	home	613	792-0000	1
2	work	613	896-1234	1
3	work	416	123-4444	2

### 36.0.2 Example of a OneToMany relationship and inverse ManyToOne annotations

```
@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private long id;
    ...
    @OneToMany(mappedBy="owner")
    private List<Phone> phones;
    ...
}

@Entity
public class Phone {
    @Id
    private long id;
    ...
    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="OWNER_ID")
    private Employee owner;
    ...
}
```

---

2 <https://java.sun.com/javase/5/docs/api/javax/persistence/OneToMany.html>

### 36.0.3 Example of a OneToMany relationship and inverse ManyToOne XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <one-to-many name="phones" target-entity="org.acme.Phone"
      mapped-by="owner"/>
  </attributes>
</entity>

<entity name="Phone" class="org.acme.Phone" access="FIELD">
  <attributes>
    <id name="id"/>
    <many-to-one name="owner" fetch="LAZY">
      <join-column name="OWNER_ID"/>
    </many-to-one>
  </attributes>
</entity>
```

Note this @OneToMany mapping requires an inverse @ManyToOne mapping to be complete, see ManyToOne<sup>3</sup>.

### 36.0.4 Getters and Setters

As the relationship is bi-directional so as the application updates one side of the relationship, the other side should also get updated, and be in synch. In JPA, as in Java in general it is the responsibility of the application, or the object model to maintain relationships. If your application adds to one side of a relationship, then it must add to the other side.

This can be resolved through add or set methods in the object model that handle both sides of the relationships, so the application code does not need to worry about it. There are two ways to go about this, you can either only add the relationship maintenance code to one side of the relationship, and only use the setter from one side (such as making the other side protected), or add it to both sides and ensure you avoid a infinite loop.

For example:

```
public class Employee {
    private List phones;
    ...
    public void addPhone(Phone phone) {
        this.phones.add(phone);
        if (phone.getOwner() != this) {
            phone.setOwner(this);
        }
    }
    ...
}

public class Phone {
    private Employee owner;
    ...
    public void setOwner(Employee employee) {
```

---

3 Chapter 33.2.2 on page 164

```
        this.owner = employee;
        if (!employee.getPhones().contains(this)) {
            employee.getPhones().add(this);
        }
    }
    ...
}
```

Some expect the JPA provider to have magic that automatically maintains relationships. This was actually part of the EJB CMP 2 specification. However the issue is if the objects are detached or serialized to another VM, or new objects are related before being managed, or the object model is used outside the scope of JPA, then the magic is gone, and the application is left figuring things out, so in general it may be better to add the code to the object model. However some JPA providers do have support for automatically maintaining relationships.

In some cases it is undesirable to instantiate a large collection when adding a child object. One solution is to not map the bi-directional relationship, and instead query for it as required. Also some JPA providers optimize their lazy collection objects to handle this case, so you can still add to the collection without instantiating it.

## 36.1 Join Table

A common mismatch between objects and relational tables is that a **OneToMany** does not require a back reference in Java, but requires a back reference foreign key in the database. Normally it is best to define the **ManyToOne** back reference in Java, if you cannot or don't want to do this, then you can use an intermediate join table to store the relationship. This is similar to a **ManyToMany** relationship, but if you add a unique constraint to the target foreign key you can enforce that it is **OneToMany**.

JPA defines a join table using the `@JoinTable`<sup>4</sup> annotation and `<join-table>` XML element. A `JoinTable` can be used on a **ManyToMany** or **OneToMany** mappings.

See also, [Unidirectional OneToMany](#)<sup>5</sup>

### 36.1.1 Example of a OneToMany using a JoinTable database

EMPLOYEE (table)

EMP_ID	FIRSTNAME	LASTNAME
1	Bob	Way
2	Sarah	Smith

EMP\_PHONE (table)

EMP_ID	PHONE_ID
1	1

---

4 <https://java.sun.com/javase/5/docs/api/javax/persistence/JoinTable.html>

5 Chapter 37.1 on page 177

1	2
2	3

PHONE (table)

ID	TYPE	AREA_CODE	P_NUMBER
1	home	613	792-0000
2	work	613	896-1234
3	work	416	123-4444

### 36.1.2 Example of a OneToMany using a JoinTable annotation

```
@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private long id;
    ...
    @OneToMany
    @JoinTable
    (
        name="EMP_PHONE",
        joinColumns={ @JoinColumn(name="EMP_ID",
referencedColumnName="EMP_ID") },
        inverseJoinColumns={ @JoinColumn(name="PHONE_ID",
referencedColumnName="ID", unique=true) }
    )
    private List<Phone> phones;
    ...
}
```

### 36.1.3 Example of a OneToMany using a JoinTable XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
    <attributes>
        <id name="id">
            <column name="EMP_ID"/>
        </id>
        <one-to-many name="phones">
            <join-table name="EMP_PHONE">
                <join-column name="EMP_ID"
referenced-column-name="EMP_ID"/>
                <inverse-join-column name="PHONE_ID"
referenced-column-name="ID" unique="true" />
            </join-table>
        </one-to-many>
    </attributes>
</entity>
```

## 36.2 See Also

- Relationships<sup>6</sup>
  - Cascading<sup>7</sup>
  - Lazy Fetching<sup>8</sup>
  - Target Entity<sup>9</sup>
  - Collections<sup>10</sup>
  - Maps<sup>11</sup>
  - Join Fetching<sup>12</sup>
  - Batch Reading<sup>13</sup>
  - Common Problems<sup>14</sup>
- ManyToOne<sup>15</sup>
- ManyToMany<sup>16</sup>

## 36.3 Common Problems

*Object not in collection after refresh.*

See Object corruption<sup>17</sup>.

---

6 Chapter 29.9 on page 123  
7 Chapter 30.2 on page 129  
8 Chapter 30.1 on page 126  
9 Chapter 30.4 on page 131  
10 Chapter 30.5 on page 132  
11 Chapter 31.2 on page 140  
12 Chapter 31.3 on page 145  
13 Chapter 31.6 on page 148  
14 Chapter 30.6 on page 135  
15 Chapter 33.2.2 on page 164  
16 Chapter 37.1.2 on page 178  
17 Chapter 31.6 on page 148

## 37 Advanced

### 37.1 Unidirectional OneToMany, No Inverse ManyToOne, No Join Table (JPA 2.0 ONLY)

JPA 1.0 does not support a unidirectional `OneToMany` relationship without a `JoinTable`. JPA 2.0 will have support for a unidirectional `OneToMany`. In JPA 2.0 a `@JoinColumn`<sup>1</sup> can be used on a `OneToMany` to define the foreign key, some JPA providers may support this already.

The main issue with an unidirectional `OneToMany` is that the foreign key is owned by the target object's table, so if the target object has no knowledge of this foreign key, inserting and updating the value is difficult. In a unidirectional `OneToMany` the source object take ownership of the foreign key field, and is responsible for updating its value.

The target object in a unidirectional `OneToMany` is an independent object, so it should not rely on the foreign key in any way, i.e. the foreign key cannot be part of its primary key, nor generally have a not null constraint on it. You can model a collection of objects where the target has no foreign key mapped, but uses it as its primary key, or has no primary key using a `Embeddable` collection mapping, see Embeddable Collections<sup>2</sup>.

If your JPA provider does not support unidirectional `OneToMany` relationships, then you will need to either add a back reference `ManyToOne` or a `JoinTable`. In general it is best to use a `JoinTable` if you truly want to model a unidirectional `OneToMany` on the database.

There are some creative workarounds to defining a unidirectional `OneToMany`. One is to map it using a `JoinTable`, but make the target table the `JoinTable`. This will cause an extra join, but work for the most part for reads, writes of course will not work correctly, so this is only a read-only solution and a hacky one at that.

#### 37.1.1 Example of a JPA 2.0 unidirectional OneToMany relationship database

EMPLOYEE (table)

EMP_ID	FIRSTNAME	LASTNAME
1	Bob	Way
2	Sarah	Smith

---

<sup>1</sup> <https://java.sun.com/javase/5/docs/api/javax/persistence/JoinColumn.html>

<sup>2</sup> Chapter 26.7 on page 97



PHONE (table)

ID	TYPE	AREA_CODE	P_NUMBER	OWNER_ID
1	home	613	792-0000	1
2	work	613	896-1234	1
3	work	416	123-4444	2

### 37.1.2 Example of a JPA 2.0 unidirectional OneToMany relationship annotations

```
@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private long id;
    ...
    @OneToMany
    @JoinColumn(name="OWNER_ID", referencedColumnName="EMP_ID")
    private List<Phone> phones;
    ...
}
```

```
@Entity
public class Phone {
    @Id
    private long id;
    ...
}
```

OneToMany<sup>3</sup> OneToMany<sup>4</sup>

---

<sup>3</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

<sup>4</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FMapping>

## 38 ManyToMany

A **ManyToMany** relationship in Java is where the source object has an attribute that stores a collection of target objects and (if) those target objects had the inverse relationship back to the source object it would also be a **ManyToMany** relationship. All relationships in Java and JPA are unidirectional, in that if a source object references a target object there is no guarantee that the target object also has a relationship to the source object. This is different than a relational database, in which relationships are defined through foreign keys and querying such that the inverse query always exists.

JPA also defines a **OneToMany**<sup>1</sup> relationship, which is similar to a **ManyToMany** relationship except that the inverse relationship (if it were defined) is a **ManyToOne** relationship. The main difference between a **OneToMany** and a **ManyToMany** relationship in JPA is that a **ManyToMany** always makes use of an intermediate relational join table to store the relationship, whereas a **OneToMany** can either use a join table, or a foreign key in target object's table referencing the source object table's primary key.

In JPA a **ManyToMany** relationship is defined through the `@ManyToMany`<sup>2</sup> annotation or the `<many-to-many>` element.

All **ManyToMany** relationships require a **JoinTable**. The **JoinTable** is defined using the `@JoinTable`<sup>3</sup> annotation and `<join-table>` XML element. The **JoinTable** defines a foreign key to the source object's primary key (`joinColumns`), and a foreign key to the target object's primary key (`inverseJoinColumns`). Normally the primary key of the **JoinTable** is the combination of both foreign keys.

### 38.0.3 Example of a ManyToMany relationship database

EMPLOYEE (table)

ID	FIRSTNAME	LASTNAME
1	Bob	Way
2	Sarah	Smith

EMP\_PROJ (table)

EMP_ID	PROJ_ID
1	1
1	2

---

<sup>1</sup> Chapter 35.1 on page 169

<sup>2</sup> <https://java.sun.com/javase/5/docs/api/javax/persistence/ManyToMany.html>

<sup>3</sup> <https://java.sun.com/javase/5/docs/api/javax/persistence/JoinTable.html>

2

1

PROJECT (table)

ID	NAME
1	GIS
2	SIG

### 38.0.4 Example of a ManyToMany relationship annotations

```
@Entity
public class Employee {
    @Id
    @Column(name="ID")
    private long id;
    ...
    @ManyToMany
    @JoinTable(
        name="EMP_PROJ",
        joinColumns={@JoinColumn(name="EMP_ID",
            referencedColumnName="ID")},
        inverseJoinColumns={@JoinColumn(name="PROJ_ID",
            referencedColumnName="ID")})
    private List<Project> projects;
    ...
}
```

### 38.0.5 Example of a ManyToMany relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id">
      <column name="EMP_ID"/>
    </id>
    <many-to-many name="projects">
      <join-table name="EMP_PROJ">
        <join-column name="EMP_ID"
          referenced-column-name="ID"/>
        <inverse-join-column name="PROJ_ID"
          referenced-column-name="ID"/>
      </join-table>
    </many-to-many>
  </attributes>
</entity>
```

## 38.1 Bi-directional Many to Many

Although a `ManyToMany` relationship is always bi-directional on the database, the object model can choose if it will be mapped in both directions, and in which direction it will be mapped in. If you choose to map the relationship in both directions, then one direction must be defined as the *owner* and the other must use the `mappedBy` attribute to define its mapping. This also avoids having to duplicate the `JoinTable` information in both places.

If the `mappedBy` is not used, then the persistence provider will assume there are two independent relationships, and you will end up getting duplicate rows inserted into the join table. If you have a conceptual bi-directional relationship, but have two different join tables in the database, then you must not use the `mappedBy`, as you need to maintain two independent tables.

As with all bi-directional relationships it is your object model's and application's responsibility to maintain the relationship in both direction. There is no *magic* in JPA, if you add or remove to one side of the collection, you must also add or remove from the other side, see object corruption<sup>4</sup>. Technically the database will be updated correctly if you only add/remove from the *owning* side of the relationship, but then your object model will be out of synch, which can cause issues.

### 38.1.1 Example of an inverse ManyToMany relationship annotation

```
@Entity
public class Project {
    @Id
    @Column(name="PROJ_ID")
    private long id;
    ...
    @ManyToMany(mappedBy="projects")
    private List<Employee> employees;
    ...
}
```

## 38.2 See Also

- Relationships<sup>5</sup>
  - Cascading<sup>6</sup>
  - Lazy Fetching<sup>7</sup>
  - Target Entity<sup>8</sup>
  - Collections<sup>9</sup>
  - Maps<sup>10</sup>
  - Join Fetching<sup>11</sup>
  - Batch Reading<sup>12</sup>
  - Common Problems<sup>13</sup>
- OneToMany<sup>14</sup>

---

4 Chapter 31.6 on page 148  
 5 Chapter 29.9 on page 123  
 6 Chapter 30.2 on page 129  
 7 Chapter 30.1 on page 126  
 8 Chapter 30.4 on page 131  
 9 Chapter 30.5 on page 132  
 10 Chapter 31.2 on page 140  
 11 Chapter 31.3 on page 145  
 12 Chapter 31.6 on page 148  
 13 Chapter 30.6 on page 135  
 14 Chapter 35.1 on page 169

## 38.3 Common Problems

### *Object not in collection after refresh.*

If you have a bi-directional `ManyToMany` relationship, ensure that you add to both sides of the relationship.

See Object corruption<sup>15</sup>.

### *Additional columns in join table.*

See Mapping a Join Table with Additional Columns<sup>16</sup>

### *Duplicate rows inserted into the join table.*

If you have a bidirectional `ManyToMany` relationship, you must use `mappedBy` on one side of the relationship, otherwise it will be assumed to be two difference relationships and you will get duplicate rows inserted into the join table.

---

<sup>15</sup> Chapter 31.6 on page 148

<sup>16</sup> Chapter 39.1 on page 183

## 39 Advanced

### 39.1 Mapping a Join Table with Additional Columns

A frequent problem is that two classes have a `ManyToMany` relationship, but the relational join table has additional data. For example if `Employee` has a `ManyToMany` with `Project` but the `PROJ_EMP` join table also has an `IS_PROJECT_LEAD` column. In this case the best solution is to create a class that models the join table. So a `ProjectAssociation` class would be created. It would have a `ManyToOne` to `Employee` and `Project`, and attributes for the additional data. `Employee` and `Project` would have a `OneToMany` to the `ProjectAssociation`. Some JPA providers also provide additional support for mapping to join tables with additional data.

Unfortunately mapping this type of model becomes more complicated<sup>1</sup> in JPA because it requires a composite primary key. The association object's `Id` is composed of the `Employee` and `Project` ids. The JPA 1.0 spec does not allow an `Id` to be used on a `ManyToOne` so the association class must have two duplicate attributes to also store the ids, and use an `IdClass`, these duplicate attributes must be kept in synch with the `ManyToOne` attributes. Some JPA providers may allow a `ManyToOne` to be part of an `Id`, so this may be simpler with some JPA providers. To make your life simpler, I would recommend adding a generated `Id` attribute to the association class. This will give the object a simpler `Id` and not require duplicating the `Employee` and `Project` ids.

This same pattern can be used no matter what the additional data in the join table is. Another usage is if you have a `Map` relationship between two objects, with a third unrelated object or data representing the `Map` key. The JPA spec requires that the `Map` key be an attribute of the `Map` value, so the *association object* pattern can be used to model the relationship.

If the additional data in the join table is only required on the database and not used in Java, such as auditing information, it may also be possible to use database triggers to automatically set the data.

#### 39.1.1 Example join table association object database

EMPLOYEE (table)

ID	FIRSTNAME	LASTNAME
1	Bob	Way
2	Sarah	Smith

---

<sup>1</sup> Chapter 23.2 on page 69

## PROJ\_EMP (table)

EMPLOYEEID	PROJECTID	IS_PROJECT_LEAD
1	1	true
1	2	false
2	1	false

## PROJECT (table)

ID	NAME
1	GIS
2	SIG

### 39.1.2 Example join table association object annotations

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToMany(mappedBy="employee")
    private List<ProjectAssociation> projects;
    ...
}

@Entity
public class Project {
    @Id
    private long id;
    ...
    @OneToMany(mappedBy="project")
    private List<ProjectAssociation> employees;
    ...
    // Add an employee to the project.
    // Create an association object for the relationship and set its
    data.
    public void addEmployee(Employee employee, boolean teamLead) {
        ProjectAssociation association = new ProjectAssociation();
        association.setEmployee(employee);
        association.setProject(this);
        association.setEmployeeId(employee.getId());
        association.setProjectId(this.getId());
        association.setIsTeamLead(teamLead);

        this.employees.add(association);
        // Also add the association object to the employee.
        employee.getProjects().add(association);
    }
}

@Entity
@Table(name="PROJ_EMP")
@IdClass(ProjectAssociationId.class)
public class ProjectAssociation {
    @Id
```

---

```

private long employeeId;
@Id
private long projectId;
@Column(name="IS_PROJECT_LEAD")
private boolean isProjectLead;
@ManyToOne
@PrimaryKeyJoinColumn(name="EMPLOYEEID", referencedColumnName="ID")
/* if this JPA model doesn't create a table for the "PROJ_EMP"
entity,
* please comment out the @PrimaryKeyJoinColumn, and use the ff:
* @JoinColumn(name = "employeeId", updatable = false, insertable =
false)
* or @JoinColumn(name = "employeeId", updatable = false, insertable
= false, referencedColumnName = "id")
*/
private Employee employee;
@ManyToOne
@PrimaryKeyJoinColumn(name="PROJECTID", referencedColumnName="ID")
/* the same goes here:
* if this JPA model doesn't create a table for the "PROJ_EMP"
entity,
* please comment out the @PrimaryKeyJoinColumn, and use the ff:
* @JoinColumn(name = "projectId", updatable = false, insertable =
false)
* or @JoinColumn(name = "projectId", updatable = false, insertable
= false, referencedColumnName = "id")
*/
private Project project;
...
}

```

```

public class ProjectAssociationId implements Serializable {

    private long employeeId;

    private long projectId;
    ...

    public int hashCode() {
        return (int)(employeeId + projectId);
    }

    public boolean equals(Object object) {
        if (object instanceof ProjectAssociationId) {
            ProjectAssociationId otherId = (ProjectAssociationId) object;
            return (otherId.employeeId == this.employeeId) &&
(otherId.projectId == this.projectId);
        }
        return false;
    }
}

```

- If the given examples won't suit your expectations, try the solution indicated in this link:

<http://giannigar.wordpress.com/2009/09/04/mapping-a-many-to-many-join-table-with-extra-col>

ManyToMany<sup>2</sup> ManyToMany<sup>3</sup>

2 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

3 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FMapping>





## 40 ElementCollection

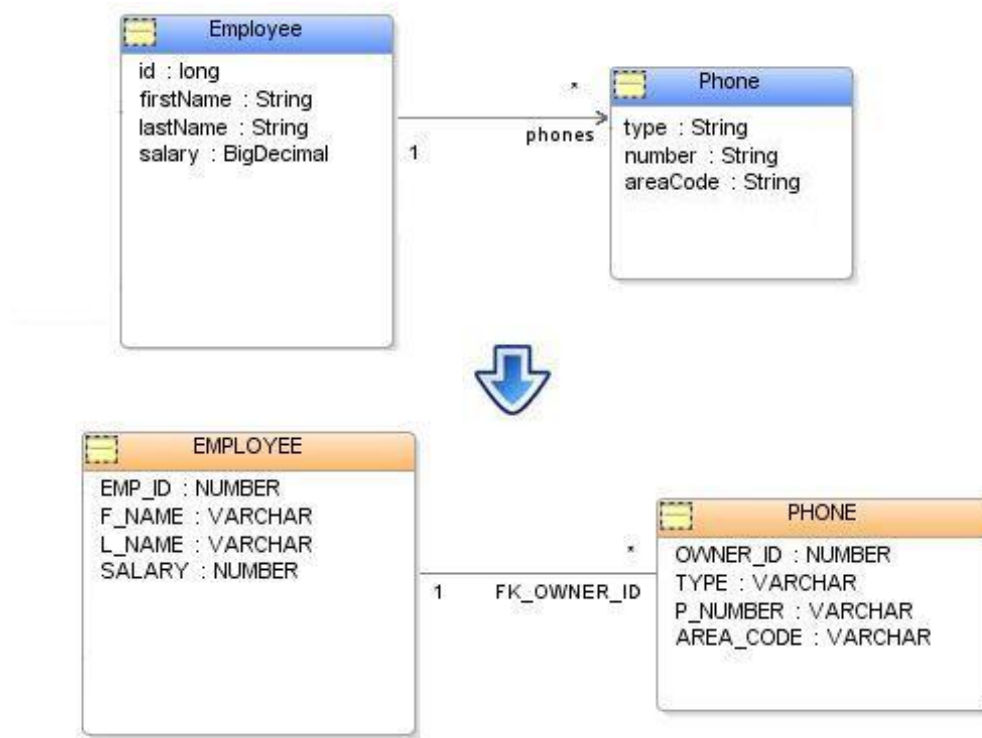


Figure 17

JPA 2.0 defines an `ElementCollection` mapping. It is meant to handle several non-standard relationship mappings. An `ElementCollection` can be used to define a one-to-many relationship to an `Embeddable` object, or a `Basic` value (such as a collection of `Strings`). An `ElementCollection` can also be used in combination with a `Map`<sup>1</sup> to define relationships where the key can be any type of object, and the value is an `Embeddable` object or a `Basic` value.

In JPA a `ElementCollection` relationship is defined through the `@ElementCollection`<sup>2</sup> annotation or the `<element-collection>` element.

The `ElementCollection` values are always stored in a separate table. The table is defined through the `@CollectionTable`<sup>3</sup> annotation or the `<collection-table>` element. The

<sup>1</sup> Chapter 31.6 on page 148

<sup>2</sup> <https://java.sun.com/javase/6/docs/api/javax/persistence/ElementCollection.html>

<sup>3</sup> <https://java.sun.com/javase/6/docs/api/javax/persistence/CollectionTable.html>

`CollectionTable` defines the table's `name` and `@JoinColumn`<sup>4</sup> or `@JoinColumns`<sup>5</sup> if a composite primary key.

## 40.1 Embedded Collections

An `ElementCollection` mapping can be used to define a collection of `Embeddable` objects. This is not a typical usage of `Embeddable` objects as the objects are not *embedded* in the source object's table, but stored in a separate collection table. This is similar to a `OneToMany`, except the target object is an `Embeddable` instead of an `Entity`. This allows collections of simple objects to be easily defined, without requiring the simple objects to define an `Id` or `ManyToOne` inverse mapping. `ElementCollection` can also override the mappings, or table for their collection, so you can have multiple entities reference the same `Embeddable` class, but have each store their dependent objects in a separate table.

The limitations of using an `ElementCollection` instead of a `OneToMany` is that the target objects cannot be queried, persisted, merged independently of their parent object. They are strictly privately-owned (dependent) objects, the same as an `Embedded` mapping. There is no `cascade` option on an `ElementCollection`, the target objects are always persisted, merged, removed with their parent. `ElementCollection` still can use a `fetch` type and defaults to `LAZY` the same as other collection mappings.

### 40.1.1 Example of an `ElementCollection` relationship database

EMPLOYEE (table)

EMP_ID	F_NAME	L_NAME	SALARY
1	Bob	Way	50000
2	Joe	Smith	35000

PHONE (table)

OWNER_ID	TYPE	AREA_CODE	P_NUMBER
1	home	613	792-0001
1	work	613	494-1234
2	work	416	892-0005

### 40.1.2 Example of an `ElementCollection` relationship annotations

```
@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
```

---

4 <https://java.sun.com/javase/5/docs/api/javax/persistence/JoinColumn.html>

5 <https://java.sun.com/javase/5/docs/api/javax/persistence/JoinColumns.html>

```

    private long id;
    ...
    @ElementCollection
    @CollectionTable(
        name="PHONE",
        joinColumns=@JoinColumn(name="OWNER_ID")
    )
    private List<Phone> phones;
    ...
}

@Embeddable
public class Phone {
    private String type;
    private String areaCode;
    @Column(name="P_NUMBER")
    private String number;
    ...
}

```

### 40.1.3 Example of an ElementCollection relationship XML

```

<entity name="Employee" class="org.acme.Employee" access="FIELD">
    <attributes>
        <id name="id">
            <column name="EMP_ID"/>
        </id>
        <element-collection name="phones">
            <collection-table name="PHONE">
                <join-column name="OWNER_ID"/>
            </collection-table>
        </element-collection>
    </attributes>
</entity>
<embeddable name="Phone" class="org.acme.Phone" access="FIELD">
    <attributes>
        <basic name="number">
            <column name="P_NUMBER"/>
        </basic>
    </attributes>
</embeddable>

```

## 40.2 Basic Collections

An `ElementCollection` mapping can be used to define a collection of `Basic` objects. The `Basic` values are stored in a separate collection table. This is similar to a `OneToMany`, except the target is a `Basic` value instead of an `Entity`. This allows collections of simple values to be easily defined, without requiring defining a class for the value.

There is no `cascade` option on an `ElementCollection`, the target objects are always persisted, merged, removed with their parent. `ElementCollection` still can use a `fetch` type and defaults to `LAZY` the same as other collection mappings.

### 40.2.1 Example of an ElementCollection relationship to a basic value database

EMPLOYEE (table)

EMP_ID	F_NAME	L_NAME	SALARY
1	Bob	Way	50000
2	Joe	Smith	35000

PHONE (table)

OWNER_ID	PHONE_NUMBER
1	613-792-0001
1	613-494-1234
2	416-892-0005

### 40.2.2 Example of a ElementCollection relationship to a basic value annotations

```
@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private long id;
    ...
    @ElementCollection
    @CollectionTable(
        name="PHONE",
        joinColumns=@JoinColumn(name="OWNER_ID")
    )
    @Column(name="PHONE_NUMBER")
    private List<String> phones;
    ...
}
```

### 40.2.3 Example of a ElementCollection relationship to a basic value XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id">
      <column name="EMP_ID"/>
    </id>
    <element-collection name="phones">
      <column name="PHONE_NUMBER"/>
      <collection-table name="PHONE">
        <join-column name="OWNER_ID"/>
      </collection-table>
    </element-collection>
  </attributes>
</entity>
```

## 40.3 See Also

- Relationships<sup>6</sup>
  - Lazy Fetching<sup>7</sup>
  - Target Entity<sup>8</sup>
  - Collections<sup>9</sup>
  - Maps<sup>10</sup>
  - Join Fetching<sup>11</sup>
  - Batch Reading<sup>12</sup>
  - Common Problems<sup>13</sup>
- OneToMany<sup>14</sup>
- ManyToMany<sup>15</sup>
- Embeddables<sup>16</sup>

## 40.4 Common Problems

### *Primary keys in CollectionTable*

The JPA 2.0 specification does not provide a way to define the `Id` in the `Embeddable`. However, to delete or update a element of the `ElementCollection` mapping, some unique *key* is normally required. Otherwise, on every update the JPA provider would need to delete everything from the `CollectionTable` for the `Entity`, and then insert the values back. So, the JPA provider will most likely assume that the combination of all of the fields in the `Embeddable` are unique, in combination with the foreign key (`JoinColumn(s)`). This however could be inefficient, or just not feasible if the `Embeddable` is big, or complex.

Some JPA providers may allow the `Id` to be specified in the `Embeddable`, to resolve this issue. Note in this case the `Id` only needs to be unique for the collection, not the table, as the foreign key is included. Some may also allow the `unique` option on the `CollectionTable` to be used for this. Otherwise, if your `Embeddable` is complex, you may consider making it an `Entity` and use a `OneToMany` instead.

`ElementCollection`<sup>17</sup> `ElementCollection`<sup>18</sup>

---

<sup>6</sup> Chapter 29.9 on page 123

<sup>7</sup> Chapter 30.1 on page 126

<sup>8</sup> Chapter 30.4 on page 131

<sup>9</sup> Chapter 30.5 on page 132

<sup>10</sup> Chapter 31.2 on page 140

<sup>11</sup> Chapter 31.3 on page 145

<sup>12</sup> Chapter 31.6 on page 148

<sup>13</sup> Chapter 30.6 on page 135

<sup>14</sup> Chapter 33.2.2 on page 164

<sup>15</sup> Chapter 37.1.2 on page 178

<sup>16</sup> Chapter 24.2.4 on page 87

<sup>17</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

<sup>18</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FMapping>



# 41 Advanced Topics

## 41.1 Events

A event is a hook into a system that allows the execution of some code when the event occurs. Events can be used to extend, integrate, debug, audit or monitor a system.

JPA defines several events for the persistent life-cycle of `Entity` objects. JPA events are defined through annotations or in the `orm.xml`. Any method of a persistent class can be annotated with an event annotation to be called for all instances of that class. An event listener can also be configured for a class using the `EntityListeners`<sup>1</sup> annotation or `<entity-listeners>` XML element. The specified listener class does not need to implement any interface (JPA does not use the Java event model), it only needs to annotate its methods with the desired event annotation.

JPA defines the following events:

- `PostLoad`<sup>2</sup> - Invoked after an `Entity` is loaded into the persistence context (`EntityManager`), or after a `refresh` operation.
- `PrePersist`<sup>3</sup> - Invoked before the `persist` operation is invoked on an `Entity`. Also invoked on `merge` for new instances, and on cascade of a `persist` operation. The Id of the object may not have been assigned, and code be assigned by the event.
- `PostPersist`<sup>4</sup> - Invoked after a new instance is persisted to the database. This occurs during a `flush` or `commit` operation after the database `INSERT` has occurred, but before the transaction is committed. It does not occur during the `persist` operation. The Id of the object should be assigned.
- `PreUpdate`<sup>5</sup> - Invoked before an instance is updated in the database. This occurs during a `flush` or `commit` operation after the database `UPDATE` has occurred, but before the transaction is committed. It does not occur during the `merge` operation.
- `PostUpdate`<sup>6</sup> - Invoked after an instance is updated in the database. This occurs during a `flush` or `commit` operation after the database `UPDATE` has occurred, but before the transaction is committed. It does not occur during the `merge` operation.
- `PreRemove`<sup>7</sup> - Invoked before the `remove` operation is invoked on an `Entity`. Also invoked for cascade of a `remove` operation. It is also invoked during a `flush` or `commit` for `orphanRemoval` in JPA 2.0.

---

1 <http://download.oracle.com/javaee/6/api/javax/persistence/EntityListeners.html>  
2 <http://download.oracle.com/javaee/6/api/javax/persistence/PostLoad.html>  
3 <http://download.oracle.com/javaee/6/api/javax/persistence/PrePersist.html>  
4 <http://download.oracle.com/javaee/6/api/javax/persistence/PostPersist.html>  
5 <http://download.oracle.com/javaee/6/api/javax/persistence/PreUpdate.html>  
6 <http://download.oracle.com/javaee/6/api/javax/persistence/PostUpdate.html>  
7 <http://download.oracle.com/javaee/6/api/javax/persistence/PreRemove.html>



- `PostRemove`<sup>8</sup> - Invoked after an instance is deleted from the database. This occurs during a `flush` or `commit` operation after the database `DELETE` has occurred, but before the transaction is committed. It does not occur during the `remove` operation.

#### 41.1.1 Example of Entity event annotations

```
@Entity
public class Employee {
    @Id
    private String uid;
    @Basic
    private Calendar lastUpdated;
    ...

    @PrePersist
    public void prePersist() {
        this.uid = UIDGenerator.newUUID();
        this.lastUpdated = Calendar.getInstance();
    }

    @PreUpdate
    public void preUpdate() {
        this.lastUpdated = Calendar.getInstance();
    }
}
```

#### 41.1.2 Example of EntityListener event annotations

```
@Entity
@EventListeners(EmployeeEventListener.class)
public class Employee {
    @Id
    private String uid;
    @Basic
    private Calendar lastUpdated;
    ...
}

public class EmployeeEventListener {
    @PrePersist
    public void prePersist(Object object) {
        Employee employee = (Employee)object;
        employee.setUID(UIDGenerator.newUUID());
        employee.setLastUpdated(Calendar.getInstance());
    }

    @PreUpdate
    public void preUpdate(Object object) {
        Employee employee = (Employee)object;
        employee.setLastUpdated(Calendar.getInstance());
    }
}
```

---

<sup>8</sup> <http://download.oracle.com/javaee/6/api/javax/persistence/PostRemove.html>

### 41.1.3 Example of Entity event xml

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <pre-persist method-name="prePersist"/>
  <pre-update method-name="preUpdate"/>
  <attributes>
    <id name="uid"/>
  </attributes>
</entity>
```

### 41.1.4 Example of EntityListener event xml

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <entity-listeners>
    <entity-listener class="org.acme.EmployeeEventListener">
      <pre-persist method-name="prePersist"/>
      <pre-update method-name="preUpdate"/>
    </entity-listener>
  </entity-listeners>
  <attributes>
    <id name="uid"/>
  </attributes>
</entity>
```

### 41.1.5 Default Entity Listeners

It is also possible to configure a default Entity listener. This listener will receive events for all of the Entity classes in the persistence unit. Default listeners can only be defined through XML.

If a default Entity listener is defined, and a class wants to define its own listener, or does not want the default listener, this can be disabled using the `ExcludeDefaultListeners`<sup>9</sup> annotation or `<exclude-default-listeners>` XML element.

### Example default Entity listener xml

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
orm_2_0.xsd">
  <persistence-unit-metadata>
    <persistence-unit-defaults>
      <entity-listeners>
        <entity-listener class="org.acme.ACMEEventListener">
          <pre-persist method-name="prePersist"/>
          <pre-update method-name="preUpdate"/>
        </entity-listener>
      </entity-listeners>
    </persistence-unit-defaults>
```

<sup>9</sup> <http://download.oracle.com/javase/6/api/javax/persistence/ExcludeDefaultListeners.html>

```
</persistence-unit-metadata>  
</entity-mappings>
```

### 41.1.6 Events and Inheritance

Entity listeners are inherited. If a subclass does not wish to inherit a superclass Entity listener, then it must define the `ExcludeSuperclassListeners`<sup>10</sup> annotation or `<exclude-superclass-listeners>` XML element.

### 41.1.7 Events and Embeddables

JPA does not define any events for `Embeddables`. Some JPA providers may allow defining events for `Embeddable` objects.

### 41.1.8 Extended Events

The JPA events are only defined for the Entity life-cycle. There are no `EntityManager` events, or system level events.

Some JPA providers may provide additional events.

`TopLink`<sup>11</sup> / `EclipseLink`<sup>12</sup> : Provide an extended event mechanism. Additional `Entity` level events are defined through the `DescriptorEventListener` API. A session level event mechanism is also provided through the `SessionEventListener` API. The event objects also provide additional information including the database row and set of object changes.

## 41.2 Views

A database `VIEW` is a virtual view of a table or query. Views are useful for hiding the complexity of a table, set of tables, or data-set.

In JPA you can map to a `VIEW` the same as a table, using the `@Table` annotation. You can then map each column in the view to your object's attributes. Views are normally read-only, so object's mapping to views are normally also read-only. In most databases views can also be updatable depending on how complex to query is that they encapsulate. Even for complex queries database triggers can normally be used to update into the view.

Views can often be used in JPA to workaround mapping limitations. For example if a table join is not supported by JPA, or a database function is desired to be called to transform data, this can normally be done inside a view, so JPA can just map to the simplified data.

Using views does require database expertise, and the definition of views can be database dependent.

---

<sup>10</sup> <http://download.oracle.com/javaee/6/api/javax/persistence/ExcludeSuperclassListeners.html>

<sup>11</sup> Chapter 11 on page 25

<sup>12</sup> Chapter 10 on page 23

## 41.3 Interfaces

Interfaces can be used for two main purposes in a model. The first is as a public interface that defines the public API for the class. The second is as a common type that allows multiple distinct implementers.

### 41.3.1 Public Interfaces

If you have a public interface in JPA, you just need to map the implementation class and are fine for the most part. One issue is that you need to use the implementation class for queries, as JPA does not know about the interface. For JPQL the default alias is also the implementation class, but you could redefine this to be the public interface by setting the `name` of the `Entity` to be the public interface.

Some JPA providers allow interfaces to be defined.

TopLink<sup>13</sup> / EclipseLink<sup>14</sup> : Support defining and querying on public interfaces using a `DescriptorCustomizer` and the `InterfacePolicy`.

#### Example public interface alias

```
@Entity(name="Employee")
public class EmployeeImpl {
    ...
}
```

### 41.3.2 Interface Types

If you have a common interfaces with multiple distinct implementers, this can have some issues. If you use the interface to define a variable relationship, then this is difficult to map. JPA has no direct support for interfaces or variable relationships. You could change the interface to be a `abstract` class, then use `TABLE_PER_CLASS` inheritance to map it. Otherwise, you could split the relationship into multiple relationships, one per each implementer, or you could just remove the relationship and query for the related objects instead. Querying on an interface is also difficult, you would need to query on each implementer of the interface, and then union the results in memory.

Some JPA providers have support for mapping interfaces, and for variable relationships.

TopLink<sup>15</sup> / EclipseLink<sup>16</sup> : Support mapping interfaces using a `SessionCustomizer` and a `RelationalDescriptor` and `InterfacePolicy`. Variable relationships can be defined using the `@VariableOneToOne` annotation or XML.

---

13 Chapter 11 on page 25

14 Chapter 10 on page 23

15 Chapter 11 on page 25

16 Chapter 10 on page 23

## 41.4 Stored Procedures

A stored procedure is a procedure or function that resides on the database. Stored procedures are typically written in some database specific language that is similar to SQL, such as PL/SQL on Oracle. Some databases such as Oracle also support stored procedures written in Java.

Stored procedures can be useful to perform batch data processing tasks. By writing the task in the database, it avoids the cost of sending the data to and from the database client, so can operate much more efficiently. Stored procedures can also be used to access database specific functionality that can only be accessed on the server. Stored procedures can also be used if strict security requirements as required, to avoid giving users access to the raw tables or unverified SQL operations. Some legacy application have also been written in database procedural languages, and need to be integrated with.

The disadvantages of using stored procedures is they are less flexible than using SQL, and require developing and maintaining functionality that is often written in a different language than the application developers may be used to, and difficult to develop and debug, and typically using a limited procedural programming language. There is also a general misconception that using stored procedures will improve performance, in that if you put the same SQL the application is executing inside a stored procedure it will somehow become faster. This is a false, and normally the opposite is true, as stored procedures restrict the dynamic ability of the persistence layer to optimize data retrieval. Stored procedures only improve performance when they use more optimal SQL than the application, typically when they perform an entire task on the database. To achieve optimal performance from SQL generated in an application you must use prepared statements - otherwise the database will have to create a new execution plan each time you submit a query.

JPA does not have any direct support for stored procedures. Some types of stored procedures can be executed in JPA through using native queries. Native queries in JPA allow any SQL that returns nothing, or returns a database result set to be executed. The syntax to execute a stored procedure depends on the database. JPA does not support stored procedures that use **OUTPUT** or **INOUT** parameters. Some databases such as DB2, Sybase and SQL Server allow for stored procedures to return result sets. Oracle does not allow results sets to be returned, only **OUTPUT** parameters, but does define a **CURSOR** type that can be returned as an **OUTPUT** parameter. Oracle also supports stored functions, that can return a single value. A stored function can normally be executed using a native SQL query by selecting the function value from the Oracle **DUAL** table.

Some JPA providers have extended support for stored procedures, some also support overriding any CRUD operation for an **Entity** with a stored procedure or custom SQL. Some JPA providers have support for **CURSOR OUTPUT** parameters.

TopLink<sup>17</sup> / EclipseLink<sup>18</sup> : Support stored procedures and stored functions using the **@NamedStoredProcedureQuery**, **@NamedStoredFunctionQuery** annotations or XML, or the **StoredProcedureCall**, **StoredFunctionCall** classes. Overriding any CRUD operation for a class or relationship are also supported using a **DescriptorCustomizer** and the

---

<sup>17</sup> Chapter 11 on page 25

<sup>18</sup> Chapter 10 on page 23

DescriptorQueryManager class. IN, OUT, INOUT, and CURSOR OUTPUT parameters are supported.

### Example executing a stored procedure on Oracle

```
EntityManager em = getEntityManager();
Query query = em.createNativeQuery("BEGIN VALIDATE_EMP(P_EMP_ID=>?);
  END;");
query.setParameter(1, empId);
query.executeUpdate();
```

#### 41.4.1 PL/SQL Stored Procedures

In Oracle stored procedures are typically written in Oracle's PL/SQL language. PL/SQL in Oracle supports some additional data-types, that Oracle does not support through SQL or JDBC. These include types such as `BOOLEAN`, `TABLE` and `RECORD`. Accessing these types or procedures is difficult from Java, as these types are not supported by JDBC. One workaround is to wrap the PL/SQL stored procedures with normal stored procedures that transform the PL/SQL types to standard SQL/JDBC types, such as `INT`, `VARRAY (Array)`, and `OBJECT TYPE (Struct)`. Some JPA providers have extended support for calling PL/SQL stored procedures.

TopLink<sup>19</sup> / EclipseLink<sup>20</sup> : Support PL/SQL stored procedures and functions using the `@NamedPLSQLStoredProcedureQuery`, `@NamedPLSQLStoredFunctionQuery` annotations or XML, or the `PLSQLStoredProcedureCall`, `PLSQLStoredFunctionCall` classes.

## 41.5 Structured Object-Relational Data Types

Back in the hay day of object-oriented databases (OODBMS) many of the relational database vendors decided to added object-oriented concepts to relational data. These new *hybrid* databases were called Object-Relational in that they could store both object and relational data. These object-relational data-types were standardized as part of SQL3 and support was added for them from Java in the JDBC 2.0 API. Although there was lots of hype around the new forms of data, object-relational data never caught on much, as people seemed to prefer their standard relational data. I would not normally recommend using object-relational data, as relational data is much more standard, but if you have really complex data, it may be something to investigate.

Some common object-relational database features include:

- Object types (structures)
- Arrays and array types
- Nested tables
- Inheritance

---

<sup>19</sup> Chapter 11 on page 25

<sup>20</sup> Chapter 10 on page 23

- Object ids (OIDs)
- Refs

Databases that support object-relational data include:

- Oracle
- DB2
- PostgreSQL

The basic model allows you to define Structs or Object-types to represent your data, the structures can have nested structures, arrays of basic data or other structures, and refs to other structures. You can then store a structure in a normal relational table column, or create a special table to store the structures directly. Querying is basic SQL, with a few extensions to handle traversing the special types.

JPA does not support object-relational data-types, but some JPA providers may offer some support.

TopLink<sup>21</sup> / EclipseLink<sup>22</sup> : Support object-relational data-types through their `@Struct`, `@Structure`, `@Array` annotations and XML, or their `ObjectRelationalDataTypeDescriptor` and mapping classes. Custom support is also offered for Oracle spatial database JGeometry structures and other structured data-types using the `@StructConverter` annotation or XML.

See also,

- Complex data stored procedures (Blog)<sup>23</sup>

## 41.6 XML Data Types

With the advent of XML databases, many relational database decided to add enhanced XML support. Although it was always possible to store XML in a relational database just using a `VARCHAR` or `CLOB` column, having the database aware of the XML data does have its advantages. The main advantage is databases that offer XML support allow querying of the XML data using XPath or XQuery syntax. Some databases also allow the XML data to be stored more efficiently than in Lob storage.

Databases with XML support include:

- Oracle (XDB)
- DB2
- PostgreSQL

JPA has no extended support for XML data, although it is possible to store an XML String into the database, just mapped as a `Basic`. Some JPA provider may offer extended XML data support. Such as query extensions, or allow mapping an XML DOM.

---

<sup>21</sup> Chapter 11 on page 25

<sup>22</sup> Chapter 10 on page 23

<sup>23</sup> <http://ronaldoblanc.blogspot.com/2011/11/jpa-eclipselink-e-stored-procedures-com.html>

If you wish to map the XML data into objects, you could make use of the JAXB specification. You may even be able to integrate this with your JPA objects.

TopLink<sup>24</sup> / EclipseLink<sup>25</sup> : Support Oracle XDB XMLType columns using their `DirectToXMLTypeMapping`. XMLTypes can be mapped either as String or as an XML DOM (Document). Query extensions are provided for XPath queries within `Expression` queries. EclipseLink also includes a JAXB implementation for object-XML mapping.

## 41.7 Filters

Some times it is desirable to filter some of the contents of a table from all queries. This is normally because the table is shared, either by multiple types, applications, tenants, or districts, and the JPA application is only interested in a subset of the rows. It may also be that the table includes historical or archive rows that should be ignored by the JPA application.

JPA does not provide any specific support for filtering data, but there are some options available. Inheritance can be used to include a type check on the rows for a class. For example, if you had a `STATUS` column in an `EMPLOYEE` table, you could define an `Employee` and a `CurrentEmployee` subclass whose discriminator `STATUS` was `ACTIVE`, and always use `CurrentEmployee` in the application. Similarly you could define an `ACMEEmployee` subclass that used the `TENANT` column as its class discriminator of value `ACME`. Another solution is to use database views to filter the data and map the entities to the views.

These solutions do not work with dynamic filtering, where the filter criteria parameters are not know until runtime (such as tenant, or district). Also complex criteria cannot be modeled through inheritance, although database views should still work. One solution is to always append the criteria to any query, such as appending a JPQL string, or JPA Criteria in the application code.

Virtual Private Database (VPD) support may also provide a solution. Some databases such as Oracle support VPD, allow context based filtering of rows based on the connected user or proxy certificate.

Some JPA providers have specific support for filtering data.

TopLink<sup>26</sup> / EclipseLink<sup>27</sup> : Support filtering data through their `@AdditionalCriteria` annotation and XML. This allows an arbitrary JPQL fragment to be appended to all queries for the entity. The fragment can contain parameters that can be set through persistence unit or context properties at runtime. Oracle VPD is also supported, include Oracle proxy authentication and isolated data.

---

<sup>24</sup> Chapter 11 on page 25

<sup>25</sup> Chapter 10 on page 23

<sup>26</sup> Chapter 11 on page 25

<sup>27</sup> Chapter 10 on page 23



## 41.8 History

A common desire in database applications is to maintain a record and history of the database changes. This can be used for tracking and auditing purposes, or to allow undoing of changes, or to keep a record of the system's data over time. Many database have auditing functionality that allows some level of tracking changes made to the data. Some databases such as Oracle's *Flashback* feature allow the automatic tracking of history at the row level, and even allow querying on past versions of the data.

It is also possible for an application to maintain its own history through its data model. All that is required is to add a **START** and **END** timestamp column to the table. The *current* row is then the one in which the **END** timestamp is null. When a row is inserted its **START** timestamp will be set to the current time. When a row is updated, instead of updating the row a new row will be inserted with the same id and data, but a different **START** timestamp, and the old row will be updated to set its **END** timestamp to the current time. The primary key of the table will need to have the **START** timestamp added to it.

History can also be used to avoid deletion. Instead of deleting a row, the **END** timestamp can just be set to the current time. Another solution is to add a **DELETED** boolean column to the table, and record when a row is deleted.

The history data could either be stored in-place in the altered table, or the table could be left to only contain the current version of the data, and a mirror history table could be added to store the data. In the mirror case, database triggers could be used to write to the history table. For the in-place case a database view could be used to give a view of the table as of the current time.

To query the current data from a history table, any query must include the clause where the **END** is **NULL**. To query as of a point in time, the where clause must include where the point in time is between the **START** and **END** timestamps.

JPA does not define any specific history support.

Oracle flashback can be used with JPA, but any queries for historical data will need to use native SQL.

If a mirror history table is used with triggers, JPA can still be used to query to current data. A subclass or sibling class could also be mapped to the history table to allow querying of history data.

If a database view is used, the JPA could be used by mapping the **Entity** to the view.

If a history table is used JPA could still be used to map to the table, and a **start** and **end** attribute could be added to the object. Queries for the current data could append the current time to the query. Relationships are more difficult, as JPA requires relationships to be by primary key, and historical relationships would not be.

Some JPA providers have support for history.

TopLink<sup>28</sup> / EclipseLink<sup>29</sup> : Support Oracle flashback querying as well as application specific history. Historical queries can be defined using the query hint "eclipselink.history.as-of" or Expression queries. Automatic tracking of history is also supported using the HistoryPolicy API that supports maintaining and querying a mirror history table.

## 41.9 Logical Deletes

### 41.10 Auditing

See, Auditing and Security<sup>30</sup>.

### 41.11 Replication

Data replication can be used to backup data, for fault tolerance and fail-over, or for load balancing and scaling the database.

For replication, changes are written to multiple databases, either by the application, JPA provider, or database back-end. For fail-over, if one of the databases goes down, the other can be used without loss of data or application downtime. For load-balancing, read requests can be load balanced across the replicated databases to reduce the load on each database, and improve the scalability of the application.

Most enterprise database support some form of automatic backup or replication. Clustered database such as Oracle RAC also allow for load balancing, fail-over and high availability. If your database supports replication or clustering, then it is normally transparent to JPA. A specialize DataSource (such as Oracle UCP, or WebLogic GridLink) may need to be used to handle load-balancing and fail-over.

JPA does not define any specific support for data replication, but some JPA provider provide replication support. If your database does not support replication, you can implement it yourself through having multiple persistence units, and persisting and merging your objects to both databases.

TopLink<sup>31</sup> / EclipseLink<sup>32</sup> : Support replication, load-balancing and fail-over. Replication and load balancing is supported through EclipseLink's partitioning support using the @ReplicationPartitioning, and @RoundRobinPartitioning annotations and XML.

---

28 Chapter 11 on page 25

29 Chapter 10 on page 23

30 [http://en.wikibooks.org/wiki/Java\\_Persistence%2FAuditing\\_and\\_Security](http://en.wikibooks.org/wiki/Java_Persistence%2FAuditing_and_Security)

31 Chapter 11 on page 25

32 Chapter 10 on page 23

## 41.12 Partitioning

Data partitioning can be used to scale an application across multiple database machines, or with a clustered database such as Oracle RAC.

Partitioning splits your data across each of the database nodes. There is horizontal partitioning, and vertical partitioning. Vertical partitioning is normally the easiest to implement. You can just put half of your classes on one database, and the other half on another. Ideally the two sets would be isolated from each other and not have any cross database relationships. This can be done directly in JPA, by having two different persistence units, one for each database.

For horizontal partitioning you need to split your data across multiple database nodes. Each database node will have the same tables, but each node's table will only store part of the data. You can partition the data by the data values, such as range partitioning, value partitioning, hash partitioning, or even round robin. JPA does not define any data partitioning support, so you either need to define a different class per partition, or use JPA vendor specific functionality.

TopLink<sup>33</sup> / EclipseLink<sup>34</sup> : Support both horizontal and vertical data partitioning. Hash, value, range, pinned and custom partitioning is supported at the Session, Entity, and Query level. Partitioning is support through the `@Partitioning`, `@HashPartitioning`, `@RangePartitioning`, `@ValuePartitioning`, `@PinnedPartitioning`, and `@Partitioned` annotations and XML.

See also,

- Data Partitioning - Scaling the Database (Blog)<sup>35</sup>

## 41.13 Data Integration

## 41.14 NoSQL (and EIS, legacy, XML, and non-relational data)

See, NoSQL<sup>36</sup>

---

<sup>33</sup> Chapter 11 on page 25

<sup>34</sup> Chapter 10 on page 23

<sup>35</sup> <http://java-persistence-performance.blogspot.com/2011/05/data-partitioning-scaling-database.html>

<sup>36</sup> [http://en.wikibooks.org/wiki/Java\\_Persistence%2FNoSQL](http://en.wikibooks.org/wiki/Java_Persistence%2FNoSQL)

## 41.15 Multi-Tenancy

## 41.16 Dynamic Data

Advanced Topics<sup>37</sup>

---

<sup>37</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>



## 42 Runtime

Once you have mapped your object model the second step in persistence development is to access and process your objects from your application, this is referred to as the runtime usage of persistence. Various persistence specifications have had various runtime models. The most common model is to have a runtime API; a runtime API typically will define API for connecting to a data-source, querying and transactions.



## 43 Entity Manager

JPA provides a runtime API defined by the `javax.persistence`<sup>1</sup> package. The main runtime class is the `EntityManager`<sup>2</sup> class. The `EntityManager` provides API for creating queries, accessing transactions, and finding, persisting, merging and deleting objects. The JPA API can be used in any Java environment including JSE and JEE.

An `EntityManager` can be created through an `EntityManagerFactory`<sup>3</sup>, or can be *injected* into an instance variable in an EJB SessionBean, or can be looked up in JNDI in a JEE server.

JPA is used differently in Java Standard Edition (JSE) versus Java Enterprise Edition (JEE).

### 43.1 Java Standard Edition

In JSE an `EntityManager` is accessed from the JPA `Persistence`<sup>4</sup> class through the `createEntityManagerFactory` API. The *persistent unit name* is passed to the `createEntityManagerFactory`, this is the name given in the persistence unit's `persistence.xml` file. All JSE JPA applications must define a `persistence.xml` file. The file defines the persistence unit including the name, classes, orm files, datasource, vendor specific properties.

JPA 1.0 does not define a standard way of specifying how to connect to the database in JSE. Each JPA provider defines their own persistence properties for setting the JDBC driver manager class, URL, user and password. JPA has a standard way of setting the `DataSource` JNDI name, but this is mainly used in JEE.

JPA 2.0 does define standard persistence unit properties for connecting to JDBC in JSE. These include, `"javax.persistence.jdbc.driver"`, `"javax.persistence.jdbc.url"`, `javax.persistence.jdbc.user`, `javax.persistence.jdbc.password`.

The JPA application is typically required to be packaged into a persistence unit jar file. This is a normal jar, that has the `persistence.xml` file in the `META-INF` directory. Typically a JPA provider will require something special be done in JSE to enable certain features such as lazy fetching, such as static weaving (byte-code processing) of the jar, or using a Java agent JVM option.

In JSE the `EntityManager` must be closed when your application is done with it. The life-cycle of the `EntityManager` is typically per client, or per request. The

---

1 <https://java.sun.com/javaee/5/docs/api/javax/persistence/package-summary.html>

2 <https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html>

3 <https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManagerFactory.html>

4 <https://java.sun.com/javaee/5/docs/api/javax/persistence/Persistence.html>



EntityManagerFactory can be shared among multiple threads or users, but the EntityManager should not be shared.

### 43.1.1 Example JPA 1.0 persistence.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
             persistence_1_0.xsd"
             version="1.0">
    <persistence-unit name="acme" transaction-type="RESOURCE_LOCAL">
        <!-- EclipseLink -->

        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

        <!-- Hibernate -->

        <!--provider>org.hibernate.ejb.HibernatePersistence</provider-->

        <!-- TopLink Essentials -->
        <!
--provider>oracle.toplink.essentials.PersistenceProvider</provider-->

        <!-- Apache OpenJPA -->
        <!--provide
r>org.apache.openjpa.persistence.PersistenceProviderImpl</provider-->

        <!-- DataNucleus-->

        <!--provider>org.datanucleus.jpa.PersistenceProviderImpl</provider-->

        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
            <property name="eclipselink.jdbc.driver"
            value="org.acme.db.Driver"/>
            <property name="eclipselink.jdbc.url"
            value="jdbc:acmedb://localhost/acme"/>
            <property name="eclipselink.jdbc.user" value="wile"/>
            <property name="eclipselink.jdbc.password"
            value="elenberry"/>
        </properties>
    </persistence-unit>
</persistence>
```

### 43.1.2 Example JPA 2.0 persistence.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
             persistence_2_0.xsd"
             version="2.0">
    <persistence-unit name="acme" transaction-type="RESOURCE_LOCAL">
        <!-- EclipseLink -->

        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

        <!-- Hibernate -->
```

```

<!--provider>org.hibernate.ejb.HibernatePersistence</provider-->

    <!-- TopLink Essentials -->
    <!
--provider>oracle.toplink.essentials.PersistenceProvider</provider-->

    <!-- Apache OpenJPA -->
    <!--provide
r>org.apache.openjpa.persistence.PersistenceProviderImpl</provider-->

    <!-- DataNucleus -->

<!--provider>org.datanucleus.jpa.PersistenceProviderImpl</provider-->

    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
        <property name="javax.persistence.jdbc.driver"
value="org.acme.db.Driver"/>
        <property name="javax.persistence.jdbc.url"
value="jdbc:acmedb://localhost/acme"/>
        <property name="javax.persistence.jdbc.user"
value="wile"/>
        <property name="javax.persistence.jdbc.password"
value="elenberry"/>
    </properties>
</persistence-unit>
</persistence>

```

### 43.1.3 Example of accessing an EntityManager from an EntityManagerFactory

```

EntityManagerFactory factory =
    Persistence.createEntityManagerFactory("acme");
EntityManager entityManager = factory.createEntityManager();
...
entityManager.close();

```

## 43.2 Java Enterprise Edition

In JEE the `EntityManager` or `EntityManagerFactory` can either be looked up in JNDI, or *injected* into a `SessionBean`. To look up the `EntityManager` in JNDI it must be published in JNDI such as through a `<persistence-context-ref>` in a `SessionBean`'s `ejb-jar.xml` file. To inject an `EntityManager` or `EntityManagerFactory` the annotation `@PersistenceContext` or `@PersistenceUnit` are used.

In JEE an `EntityManager` can either be *managed* (container-managed) or *non-managed* (application-managed). A managed `EntityManager` has a different life-cycle than an `EntityManager` managed by the application. A managed `EntityManager` should never be closed, and integrates with JTA transactions so local transaction cannot be used. Across each JTA transaction boundary all of the entities read or persisted through a managed `EntityManager` become detached. Outside of a JTA transaction a managed `EntityManager`'s behavior is sometimes odd, so typically should be used inside a JTA transaction.

A non-managed `EntityManager` is one that is created by the application through a `EntityManagerFactory` or directly from `Persistence`. A non-managed `EntityManager`

must be closed, and typically does not integrate with JTA, but this is possible through the `joinTransaction`<sup>5</sup> API. The entities in a non-managed `EntityManager` do not become detached after a transaction completes, and can continue to be used in subsequent transactions.

### 43.2.1 Example JEE JPA persistence.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
             persistence_1_0.xsd"
             version="1.0">
    <persistence-unit name="acme" transaction-type="JTA">
        <jta-data-source>jdbc/ACMEDataSource</jta-data-source>
    </persistence-unit>
</persistence>
```

### Example SessionBean ejb-jar.xml file with persistence context

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
         version="3.0">
    <enterprise-beans>
        <session>
            <ejb-name>EmployeeService</ejb-name>

            <business-remote>org.acme.EmployeeService</business-remote>
            <ejb-class>org.acme.EmployeeServiceBean</ejb-class>
            <session-type>Stateless</session-type>
            <persistence-context-ref>
                <persistence-context-ref-name>persistence/acme/entity-manager</persistence-context-ref-name>
                <persistence-unit-name>acme</persistence-unit-name>
            </persistence-context-ref>
            <persistence-unit-ref>
                <persistence-unit-ref-name>persistence/acme/factory</persistence-unit-ref-name>
                <persistence-unit-name>acme</persistence-unit-name>
            </persistence-unit-ref>
        </session>
    </enterprise-beans>
</ejb-jar>
```

### Example of looking up an EntityManager in JNDI from a SessionBean

```
InitialContext context = new InitialContext(properties);
EntityManager entityManager = (EntityManager)
context.lookup("java:comp/env/persistence/acme/entity-manager");
...
```

---

<sup>5</sup> Chapter 50.1 on page 257

## Example of looking up an EntityManagerFactory in JNDI from a SessionBean

```
InitialContext context = new InitialContext(properties);
EntityManagerFactory factory = (EntityManagerFactory)context.lookup("java:comp/env/persistence/acme/factory");
...
```

## Example of injecting an EntityManager and EntityManagerFactory in a SessionBean

```
@Stateless(name="EmployeeService", mappedName="acme/EmployeeService")
@Remote(EmployeeService.class)
public class EmployeeServiceBean implements EmployeeService {

    @PersistenceContext(unitName="acme")
    private EntityManager entityManager;

    @PersistenceUnit(unitName="acme")
    private EntityManagerFactory factory;
    ...
}
```

## Example of lookup an EJBContext in an Entity

(Useful for Audit<sup>6</sup>)

```
protected EJBContext getContext() {
    try {
        InitialContext context = new InitialContext();
        return (EJBContext)context.lookup("java:comp/EJBContext");
    } catch (NamingException e) {
        throw new EJBException(e);
    }
}
```

### 43.2.2 Stateless SessionBeans

### 43.2.3 Stateful SessionBeans

Runtime<sup>7</sup> Runtime<sup>8</sup>

---

<sup>6</sup> Chapter 41.10 on page 203

<sup>7</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

<sup>8</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FRuntime>



## 44 Querying

Querying is a fundamental part of persistence. Being able to persist something is not very useful without being able to query it back. There are many querying languages and frameworks; the most common query language is SQL used in relational databases.

JPA provides several querying mechanisms:

- JPQL<sup>1</sup> : (BNF<sup>2</sup>)
- Criteria API<sup>3</sup>
- Native SQL Queries<sup>4</sup>

JPA primarily uses the Java Persistence Querying Language (JPQL), which is based on the SQL language and evolved from the EJB Query Language (EJBQL). It basically provides the SQL syntax at the object level instead of at the data level. JPQL is similar in syntax to SQL and can be defined through its BNF<sup>5</sup> definition.

JPA also provides the Criteria API that allows dynamic queries to be easily built using a Java API. The Criteria API mirrors the JPQL syntax, but provides Java API for each operation/function instead of using a separate query language.

JPA provides querying through the `Query` interface, and the `@NamedQuery` and `@NamedNativeQuery` annotations and the `<named-query>` and `<named-native-query>` XML elements.

Other querying languages and frameworks include:

- SQL<sup>6</sup>
- EJBQL
- JDOQL
- EQL<sup>7</sup> (EclipseLink Query Language)
- Query By Example (QBE)
- TopLink Expressions
- Hibernate Criteria
- Object Query Language (OQL)<sup>8</sup>
- Query DSL<sup>9</sup>

---

1 <http://en.wikibooks.org/wiki/Java%20Persistence%2FJPQL>

2 Chapter 45.8 on page 232

3 <http://en.wikibooks.org/wiki/Java%20Persistence%2FCriteria>

4 Chapter 45.6 on page 229

5 Chapter 45.8 on page 232

6 <http://en.wikipedia.org/wiki/SQL>

7 [http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic\\_JPA\\_Development/Querying/JPQL#EclipseLink\\_Extensions\\_.28EQL.29](http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL#EclipseLink_Extensions_.28EQL.29)

8 <http://en.wikipedia.org/wiki/Object%20Query%20Language>

9 <http://www.querydsl.com/>

## 44.1 Named Queries

There are two main types of queries in JPA, named queries and dynamic queries. A named query is used for a static query that will be used many times in the application. The advantage of a named query is that it can be defined once, in one place, and reused in the application. Most JPA providers also pre-parse/compile named queries, so they are more optimized than dynamic queries which typically must be parsed/compiled every time they are executed. Since named queries are part of the persistence meta-data they can also be optimized or overridden in the `orm.xml` without changing the application code.

Named queries are defined through the `@NamedQuery`<sup>10</sup> and `@NamedQueries`<sup>11</sup> annotations, or `<named-query>` XML element. Named queries are accessed through the `EntityManager.createNamedQuery`<sup>12</sup> API, and executed through the `Query`<sup>13</sup> interface.

Named queries can be defined on any annotated class, but are typically defined on the `Entity` that they query for. The name of the named query must be unique for the entire persistence unit, they name is not local to the `Entity`. In the `orm.xml` named queries can be defined either on the `<entity-mappings>` or on any `<entity>`.

Named queries are typically parametrized, so they can be executed with different parameter values. Parameters are defined in JPQL using the `:<name>` syntax for named parameters, or the `?` syntax for positional parameters.

A collection of query hints can also be provided to a named query. Query hints can be used to optimize or to provide special configuration to a query. Query hints are specific to the JPA provider. Query hints are defined through the `@QueryHint`<sup>14</sup> annotation or `query-hint` XML element.

### 44.1.1 Example named query annotation

```
@NamedQuery(  
    name="findAllEmployeesInCity",  
    query="Select emp from Employee emp where emp.address.city = :city"  
    hints={@QueryHint(name="acme.jpa.batch", value="emp.address")}  
)  
public class Employee {  
    ...  
}
```

- 
- 10 <https://java.sun.com/javaee/5/docs/api/javax/persistence/NamedQuery.html>
  - 11 <https://java.sun.com/javaee/5/docs/api/javax/persistence/NamedQueries.html>
  - 12 [https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#createNamedQuery\(java.lang.String\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#createNamedQuery(java.lang.String))
  - 13 <https://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html>
  - 14 <https://java.sun.com/javaee/5/docs/api/javax/persistence/QueryHint.html>

### 44.1.2 Example named query XML

```
<entity-mappings>
  <entity name="Employee" class="org.acme.Employee" access="FIELD">
    <named-query name="findAllEmployeesInCity">
      <query>Select emp from Employee emp where emp.address.city =
:city</query>
      <hint name="acme.jpa.batch" value="emp.address"/>
    </named-query>
    <attributes>
      <id name="id"/>
    </attributes>
  </entity>
</entity-mappings>
```

### 44.1.3 Example named query execution

```
EntityManager em = getEntityManager();
Query query = em.createNamedQuery("findAllEmployeesInCity");
query.setParameter("city", "Ottawa");
List<Employee> employees = query.getResultList();
...
```

## 44.2 Dynamic Queries

Dynamic queries are normally used when the query depends on the context. For example, depending on which items in the query form were filled in, the query may have different parameters. Dynamic queries are also useful for uncommon queries, or prototyping.

JPA provides two main options for dynamic queries, JPQL and the Criteria API.

Dynamic queries can use parameters, and query hints the same as named queries.

Dynamic queries are accessed through the `EntityManager.createQuery`<sup>15</sup> API, and executed through the `Query`<sup>16</sup> interface.

### Example dynamic query execution

```
EntityManager em = getEntityManager();
Query query = em.createQuery("Select emp from Employee emp where
  emp.address.city = :city");
query.setParameter("city", "Ottawa");
query.setHint("acme.jpa.batch", "emp.address");
List<Employee> employees = query.getResultList();
...
```

<sup>15</sup> [https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#createQuery\(java.lang.String\)](https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#createQuery(java.lang.String))

<sup>16</sup> <https://java.sun.com/javase/5/docs/api/javax/persistence/Query.html>



### 44.2.1 Criteria API (JPA 2.0)

See Criteria API<sup>17</sup>.

### 44.3 JPQL

See JPQL<sup>18</sup>.

### 44.4 Parameters

Parameters are defined in JPQL using the `:` syntax, i.e. `"Select e from Employee e where e.id = :id"`. The parameter values are set on the `Query` using the `Query.setParameter`<sup>19</sup> API.

Parameters can also be defined using the `?`, mainly for native SQL queries. You can also use `?<int>`. These are positional parameters, not named parameters and are set using the `Query API` `Query.setParameter`<sup>20</sup>. The `int` is the index of the parameter in the SQL. Positional parameters start at 1 (not 0). Some JPA providers also allow the `:` syntax for native queries.

For temporal parameters (`Date`, `Calendar`) you can also pass the temporal type, depending on if you want the `Date`, `Time` or `Timestamp` from the value.

Parameters are normally basic values, but you can also reference object's if comparing on their Id, i.e. `"Select e from Employee e where e.address = :address"`, can take the `Address` object as a parameter. The parameter values are always at the object level when comparing to a mapped attribute, for example if comparing a mapped `enum` the `enum` value is used, not the database value.

Parameters are always set on the `Query`, no matter what type of query it is (JPQL, Criteria, native SQL, `NamedQuery`).

Named Parameter:

```
Query query = em.createQuery("Select e from Employee e where e.name =  
:name");  
query.setParameter("name", "Bob Smith");
```

Positional Parameter:

```
Query query = em.createNativeQuery("SELECT * FROM EMPLOYEE WHERE NAME  
= ?");  
query.setParameter(1, "Bob Smith");
```

---

17 <http://en.wikibooks.org/wiki/Java%20Persistence%2FCriteria>

18 <http://en.wikibooks.org/wiki/Java%20Persistence%2FJPQL>

19 [https://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html#setParameter\(java.lang.String,%20java.lang.Object\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html#setParameter(java.lang.String,%20java.lang.Object))

20 [https://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html#setParameter\(int,%20java.lang.Object\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html#setParameter(int,%20java.lang.Object))

## 44.5 Query Results

Normally JPA queries return your persistent `Entity` objects. The returned objects will be managed by the persistent context (`EntityManager`) and changes made to the objects will be tracked as part of the current transaction. In some case more complex queries can be built that just return data, instead of `Entity` objects, or even perform *update* or *deletion* operations.

There are three methods to execute a `Query`, each returning different results:

- `Query.getResultList`<sup>21</sup>
- `Query.getSingleResult`<sup>22</sup>
- `Query.executeUpdate`<sup>23</sup>

`getResultList` returns a `List` of the results. This is normally a `List` of `Entity` objects, but could also be a list of data, or arrays.

JPQL / SQL	Result
<code>SELECT e FROM Employee e</code>	This returns a <code>List&lt;Employee&gt;</code> (List of <code>Employee</code> objects). The objects will be managed.
<code>SELECT e.firstName FROM Employee e</code>	This returns a <code>List&lt;String&gt;</code> (List of <code>String</code> values). The data is not managed.
<code>SELECT e.firstName, e.lastName FROM Employee e</code>	This returns a <code>List&lt;Object[String, String]&gt;</code> (List of object arrays each with two <code>String</code> values). The data is not managed.
<code>SELECT e, e.address FROM Employee e</code>	This returns a <code>List&lt;Object[Employee, Address]&gt;</code> (List of object arrays each with an <code>Employee</code> and <code>Address</code> objects). The objects will be managed.
<code>SELECT EMP_ID, F_NAME, L_NAME FROM EMP</code>	This returns a <code>List&lt;Object[BigDecimal, String, String]&gt;</code> (List of object arrays each with the row data). The data is not managed.

`getSingleResult` returns the results. This is normally an `Entity` objects, but could also be data, or an object array. If the query returns nothing, an exception is thrown. This is unfortunate, as typically just returning `null` would be desired. Some JPA providers may have an option to return `null` instead of throwing an exception if nothing is returned. Also if the query returns more than a single row, and exception is also thrown. This is also unfortunate, as typically just returning the first result is desired. Some JPA providers may have an option to return the first result instead of throwing an exception, otherwise you need to call `getResultList` and get the first element.

<sup>21</sup> [https://java.sun.com/javase/5/docs/api/javax/persistence/Query.html#getResultList\(\)](https://java.sun.com/javase/5/docs/api/javax/persistence/Query.html#getResultList())

<sup>22</sup> [https://java.sun.com/javase/5/docs/api/javax/persistence/Query.html#getSingleResult\(\)](https://java.sun.com/javase/5/docs/api/javax/persistence/Query.html#getSingleResult())

<sup>23</sup> [https://java.sun.com/javase/5/docs/api/javax/persistence/Query.html#executeUpdate\(\)](https://java.sun.com/javase/5/docs/api/javax/persistence/Query.html#executeUpdate())

JPQL / SQL	Result
SELECT e FROM Employee e	This returns an <code>Employee</code> . The object will be managed.
SELECT e.firstName FROM Employee e	This returns a <code>String</code> . The data is not managed.
SELECT e.firstName, e.lastName FROM Employee e	This returns an <code>Object[String, String]</code> (object array with two <code>String</code> values). The data is not managed.
SELECT e, e.address FROM Employee e	This returns an <code>Object[Employee, Address]</code> (object array with an <code>Employee</code> and <code>Address</code> object). The objects will be managed.
SELECT EMP_ID, F_NAME, L_NAME FROM EMP	This returns an <code>Object[BigDecimal, String, String]</code> (object array with the row data). The data is not managed.

`executeUpdate` returns the database row count. This can be used for `UPDATE DELETE` JPQL queries, or any native SQL (DML or DDL) query that does not return a result.

## 44.6 Common Queries

### 44.6.1 *Joining, querying on a OneToMany relationship*

To query *all employees with a phone number in 613 area code* a *join* is used.

JPQL:

```
SELECT e FROM Employee e JOIN e.phoneNumbers p where p.areaCode =
,613,
```

Criteria:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Employee> query = cb.createQuery(Employee.class);
Root<Employee> employee = query.from(Employee.class);
Join<PhoneNumber> phone = employee.join("phoneNumbers");
query.where(cb.equal(phone.get("areaCode"), "613"));
```

### 44.6.2 *Subselect, querying all of a ManyToMany relationship*

To query *all employees whose projects are all in trouble* a *subselect* with a double negation is used.

JPQL:

```
SELECT e FROM Employee e JOIN e.projects p where NOT EXISTS (SELECT t
from Project t where p = t AND t.status <> ,In trouble,)
```

Criteria:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Employee> query = cb.createQuery(Employee.class);
Root<Employee> employee = query.from(Employee.class);
Join<Project> project = employee.join("projects");
Subquery<Project> subquery = query.subquery(Project.class);
Root<Project> subProject = query.from(Project.class);
subquery.where(cb.and(cb.equal(project, subProject),
    cb.equal(subProject.get("status"), "In trouble")));
query.where(cb.not(cb.exists(subquery)));
```

### 44.6.3 *Join fetch, read both employee and address in same query*

To query *all employees and their address* a *join fetch* is used. This selects both the employee and address data in the same query. If the join fetch was not used, the employee address would still be available, but could cause a query for each employee for its address. This reduces  $n+1$  queries to 1 query.

Join fetch:

```
SELECT e FROM Employee e JOIN FETCH e.address
```

Join fetch can also be used on collection relationships:

```
SELECT e FROM Employee e JOIN FETCH e.address JOIN FETCH e.phones
```

Outer joins can be used to avoid null and empty relationships from filtering the results:

```
SELECT e FROM Employee e LEFT OUTER JOIN FETCH e.address LEFT OUTER
JOIN FETCH e.phones
```

You can also select multiple objects in a query, but note that this does not instantiate the relationship, so accessing the relationship could still trigger another query:

```
SELECT e, a FROM Employee e, Address a WHERE e.address = a
```

### 44.6.4 *Inverse ManyToMany, all employees for a given project*

To query *all employees for a given project* where the employee project relationship is a ManyToMany.

If the relationship is bi-directional you could use:

```
Select p.employees from Project p where p.name = :name
```

If it is uni-directional you could use:

```
Select e from Employee e, Project p where p.name = :name and p member
of e.projects
```

or,

```
Select e from Employee e join employee.projects p where p.name =
:name
```

#### 44.6.5 *How to simulate casting to a subclass*

To query *all employees who have a large project with a budget greater than 1,000,000* where the employee only has a relationship to Project, not to the LargeProject subclass. JPA 1.0 JPQL does not define a cast operation (JPA 2.0 may define this), so querying on an attribute of a subclass is not obvious. This can be done indirectly however, if you add a secondary join to the subclass to the query.

```
Select e from Employee e join employee.projects p, LargeProject
  lproject where p = lproject and lproject.budget > 1000000
```

#### 44.6.6 *How to select the first element in a collection*

To query *the employees first project* for a particular employee. There are a few different ways to do this, some using straight JPQL, and some using the Query `setMaxResults` API. If a JPA 2.0 indexed list is used to map the collection, then the `INDEX` function can be used.

**setMaxResults:**

```
Query query = em.createQuery("Select e.projects from Employee e where
  e.id = :id");
query.setMaxResults(1);
```

```
Query query = em.createQuery("Select p from Employee e join
  e.projects p where e.id = :id");
query.setMaxResults(1);
```

**JPQL:**

```
Select p from Project p where p.id = (Select MAX(p2.id) from Employee
  e join e.projects p2 where e.id = :id)
```

**JPA 2.0:**

```
Select p from Employee e join e.projects p where e.id = :id and
  INDEX(p) = 1
```

#### 44.6.7 *How to order by the size of a collection*

To query *all employees ordered by the number of projects*. There are a few different ways to do this, some end up using sub selects in SQL, and some use group by. Depending on your JPA provider and database your solution may be limited to one or the other.

**Using SIZE function (uses sub-select in SQL)**

```
Select e from Employee order by SIZE(e.projects) DESC
```

**Using SIZE function, also selects the size (uses group by)**

```
Select e, SIZE(e.projects) from Employee order by SIZE(e.projects)
  DESC
```

## Using GROUP BY

```
Select e, COUNT(p) from Employee join e.projects p order by COUNT(p)  
DESC
```

## Using GROUP BY and alias

```
Select e, COUNT(p) as pcount from Employee join e.projects p order by  
pcount DESC
```



# 45 Advanced

## 45.1 Join Fetch and Query Optimization

There are several ways to optimize queries in JPA. The typical query performance issue is that an object is read first, then its related objects are read one by one. This can be optimized using `JOIN FETCH` in JPQL, otherwise by query hints specific for each JPA provider.

See,

- Join Fetching<sup>1</sup>
- Batch Fetching<sup>2</sup>

## 45.2 Timeouts, Fetch Size and other JDBC Optimizations

There are several JDBC options that can be used when executing a query. These JDBC options are not exposed by JPA, but some JPA providers may support query hints for them.

- Fetch size : Configures the number of rows to fetch from the database in each page. A larger fetch size is more efficient for large queries.
- Timeout : Instructs the database to cancel the query if its execution takes too long.

EclipseLink<sup>3</sup>/TopLink<sup>4</sup> : Provide many query hints including:

"eclipselink.jdbc.fetch-size" - Fetch size.

"eclipselink.jdbc.timeout" - Timeout.

"eclipselink.read-only" - The objects returned from the query are not managed by the persistence context, and not tracked for changes.

"eclipselink.query-type" - Defines the native type of query to use for the query.

"eclipselink.sql.hint" - Allows an SQL hint to be included in the SQL for the query.

"eclipselink.jdbc.bind-parameters" - Specifies if parameter binding should be used or not, (is used by default).

---

1 Chapter 31.3 on page 145

2 Chapter 31.4 on page 147

3 Chapter 10 on page 23

4 Chapter 11 on page 25



## 45.3 Update and Delete Queries

JPQL also allows for `UPDATE` and `DELETE` queries to be executed. This is not the recommended or normal way to modify objects in JPA. Normally in JPA you first read the object, then either modify it directly using its `set` methods to update it, or call the `EntityManager.remove()` method to delete it.

`UPDATE` and `DELETE` queries in JPQL are for performing batch updates or deletions. They allow a set of objects to be updated or deleted in a single query. These queries are useful for performing batch operations, or clearing test data.

`UPDATE` and `DELETE` queries have a `WHERE` clause the same as `SELECT` queries, and can use the same functions and operations, and traverse relationships and make use of *sub selects*. `UPDATE` and `DELETE` queries are executed using the `Query.executeUpdate()` method, and return the row count from the database. Note that some caution should be used in executing these queries in an active persistence context, as the queries may effect the objects that have already been registered in the `EntityManager`. Normally it is a good idea to `clear()` the `EntityManager` after executing the query, or to execute the query in a new `EntityManager` or transaction.

### 45.3.1 Example update query

```
UPDATE Employee e SET e.salary = e.salary + 1000 WHERE e.address.city
= :city
```

### 45.3.2 Example delete query

```
DELETE FROM Employee e WHERE e.address.city = :city
```

## 45.4 Flush Mode

Within a transaction context in JPA, changes made to the managed objects are normally not flushed (written) to the database until commit. So if a query were executed against the database directly, it would not see the changes made within the transaction, as these changes are only made in memory within the Java. This can cause issues if new objects have been persisted, or objects have been removed or changed, as the application may expect the query to return these results. Because of this JPA requires that the JPA provider performs a flush of all changes to the database before any query operation. This however can cause issues if the application is not expecting that a flush as a side effect of a query operation. If the application changes are not yet in a state to be flushed, a flush may not be desired. Flushing also can be expensive and causes the database transaction, and database locks are other resources to be held for the duration of the transaction, which can effect performance and concurrency.

JPA allows the flush mode for a query to be configured using the `FlushModeType`<sup>5</sup> enum and the `Query.setFlushMode()`<sup>6</sup> API. The flush mode is either `AUTO` the default which means flush before every query execution, or `COMMIT` which means only flush on commit. The flush mode can also be set on an `EntityManager` using the `EntityManager.setFlushMode()`<sup>7</sup> API, to affect all queries executed with the `EntityManager`. The `EntityManager.flush()`<sup>8</sup> API can be called directly on the `EntityManager` anytime that a flush is desired.

Some JPA providers also let the flush mode be configured through persistence unit properties, or offer alternatives to flushing, such as performing the query against the in memory objects.

TopLink<sup>9</sup> / EclipseLink<sup>10</sup> : Allow the auto flush to be disabled using the persistence unit property `"eclipselink.persistence-context.flush-mode"="COMMIT"`.

## 45.5 Pagination, Max/First Results

A common requirement is to allow the user to page through a large query result. Typically a web user is given the first page of  $n$  results after a query execution, and can click *next* to go to the next page, or *previous* to go back.

If you are not concerned about performance, or the results are not too big, the easiest way to implement this is to query all of the results, then access the sub-list from the result list to populate your page. However, you will then have to re-query the entire results on every page request.

One simple solution is to store the query results in a *stateful SessionBean* or an *http session*. This means the initial query make take a while, but paging will be fast. Some JPA providers also support the caching of query results, so you can cache the results in your JPA providers cache and just re-execute the query to obtain the cached results.

If the query result is quite large, then another solution may be required. JPA provides the `Query` API `setFirstResult`<sup>11</sup>, `setMaxResults`<sup>12</sup> to allow paging through a large query result. The `maxResults` can also be used as a safeguard to avoid letting users execute queries that return too many objects.

How these query properties are implemented depends on the JPA provider and database. JDBC allows the `maxResults` to be set, and most JDBC drivers support this, so it will normally work for most JPA providers and most databases. Support for `firstResult` can be less guaranteed to be efficient, as it normally requires database specific SQL. There is no

---

5 <https://java.sun.com/javaee/5/docs/api/javax/persistence/FlushModeType.html>  
 6 [https://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html#setFlushMode\(javax.persistence.FlushModeType\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html#setFlushMode(javax.persistence.FlushModeType))  
 7 [https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#setFlushMode\(javax.persistence.FlushModeType\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#setFlushMode(javax.persistence.FlushModeType))  
 8 [https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#flush\(\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#flush())  
 9 Chapter 11 on page 25  
 10 Chapter 10 on page 23  
 11 [https://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html#setFirstResult\(int\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html#setFirstResult(int))  
 12 [https://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html#setMaxResults\(int\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html#setMaxResults(int))

standard SQL for pagination, so whether if this is supported depends on your database, and your JPA providers support.

When performing pagination, it is also important to *order* the result. If the query does not order the result, then each subsequent query could potentially return the results in a different order, and give a different page. Also if rows are insert/deleted in between the queries, the results can be slightly different.

#### 45.5.1 Example using `firstResult`, `maxResults`

```
Query query = em.createQuery("Select e from Employee e order by  
    e.id");  
query.setFirstResult(100);  
query.setMaxResults(200);  
List<Employee> page = query.getResultList();
```

An alternative to using `firstResult` is to filter the first result in the where clause based on the order by and the value from the previous page.

#### 45.5.2 Example using `maxResults` and `order by`

```
Query query = em.createQuery("Select e from Employee e where e.id >  
    :lastId order by e.id");  
query.setParameter("lastId",  
    previousPage.get(previousPage.size()-1).getId());  
query.setMaxResults(100);  
List<Employee> nextPage = query.getResultList();
```

Another alternative is to only query the Ids, and store this result in a *stateful* `SessionBean` or an *http session*. Then query for the set of Ids for each page.

#### 45.5.3 Example using Ids and IN

```
Query query = em.createQuery("Select e.id from Employee e");  
List<Long> ids= query.getResultList();  
  
Query pageQuery = em.createQuery("Select e from Employee e where e.id  
    in :ids");  
pageQuery.setParameter("ids", ids.subList(100, 200));  
List<Employee> page = pageQuery.getResultList();
```

Pagination can also be used for server processes, or batch jobs. On the server, it is normally used to avoid using too much memory upfront, and allow processing each batch one at a time. Any of these techniques can be used, also some JPA providers support returning a database *cursor* for the query results that allows scrolling through the results.

TopLink<sup>13</sup> / EclipseLink<sup>14</sup> : Support streams and scrollable cursors through the query hints "eclipselink.cursor.scrollable" and "eclipselink.cursor", and `CursoredStream` and `ScrollableCursor` classes.

---

<sup>13</sup> Chapter 11 on page 25

<sup>14</sup> Chapter 10 on page 23

## 45.6 Native SQL Queries

Typically queries in JPA are defined through JPQL. JPQL allows the queries to be defined in terms of the object model, instead of the data model. Since developers are programming in Java using the object model, this is normally more intuitive. This also allows for data abstraction and database schema and database platform independence. JPQL supports much of the SQL syntax, but some aspects of SQL, or specific database extensions or functions may not be possible through JPQL, so native SQL queries are sometimes required. Also some developers have more experience with SQL than JPQL, so may prefer to use SQL queries. Native queries can also be used for calling some types of stored procedures or executing DML or DDL operations.

Native queries are defined through the `@NamedNativeQuery`<sup>15</sup> and `@NamedNativeQueries`<sup>16</sup> annotations, or `<named-native-query>` XML element. Native queries can also be defined dynamically using the `EntityManager.createNativeQuery()`<sup>17</sup> API.

A native query can be for a query for instances of a class, a query for raw data, an update or DML or DDL operation, or a query for a complex query result. If the query is for a class, the `resultClass` attribute of the query must be set. If the query result is complex, a Result Set Mapping<sup>18</sup> can be used.

Native queries can be parameterized, so they can be executed with different parameter values. Parameters are defined in SQL using the `?` syntax for positional parameters, JPA does not require native queries support named parameters, but some JPA providers may. For positional parameter the position starts a 1 (not 0).

A collection of query hints can also be provided to a native query. Query hints can be used to optimize or to provide special configuration to a query. Query hints are specific to the JPA provider. Query hints are defined through the `@QueryHint`<sup>19</sup> annotation or `query-hint` XML element.

### Example native named query annotation

```
@NamedNativeQuery(
    name="findAllEmployeesInCity",
    query="SELECT E.* from EMP E, ADDRESS A WHERE E.EMP_ID = A.EMP_ID
AND A.CITY = ?",
    resultClass=Employee.class
)
public class Employee {
    ...
}
```

15 <https://java.sun.com/javase/5/docs/api/javax/persistence/NamedNativeQuery.html>  
16 <https://java.sun.com/javase/5/docs/api/javax/persistence/NamedNativeQueries.html>  
17 [https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#createNativeQuery\(java.lang.String\)](https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#createNativeQuery(java.lang.String))  
18 Chapter 45.6.1 on page 230  
19 <https://java.sun.com/javase/5/docs/api/javax/persistence/QueryHint.html>

## Example native named query XML

```
<entity-mappings>
  <entity name="Employee" class="org.acme.Employee" access="FIELD">
    <named-native-query name="findAllEmployeesInCity"
      result-class="org.acme.Employee">
      <query>SELECT E.* from EMP E, ADDRESS A WHERE E.EMP_ID =
A.EMP_ID AND A.CITY = ?</query>
    </named-native-query>
    <attributes>
      <id name="id"/>
    </attributes>
  </entity>
</entity-mappings>
```

## Example native named query execution

```
EntityManager em = getEntityManager();
Query query = em.createNamedQuery("findAllEmployeesInCity");
query.setParameter(1, "Ottawa");
List<Employee> employees = query.getResultList();
...
```

## Example dynamic native query execution

```
EntityManager em = getEntityManager();
Query query = em.createNativeQuery("SELECT E.* from EMP E, ADDRESS A
WHERE E.EMP_ID = A.EMP_ID AND A.CITY = ?", Employee.class);
query.setParameter(1, "Ottawa");
List<Employee> employees = query.getResultList();
...
```

### 45.6.1 Result Set Mapping

When a native SQL query returns objects, the SQL must ensure it returns the correct data to build the `resultClass` using the correct column names as specified in the mappings. If the SQL is more complex and returns different column names, or returns data for multiple objects then a `@SqlResultSetMapping`<sup>20</sup> must be used.

`@SqlResultSetMapping` is a fairly complex annotation containing an array of `@EntityResult`<sup>21</sup> and `@ColumnResult`<sup>22</sup>. This allows multiple `Entity` objects in combination with raw data to be returned. The `@EntityResult` contains an array of `@FieldResult`<sup>23</sup>, which can be used to map the alias name used in the SQL to the column name required by the mapping. This is required if you need to return two different instances of the same class, or if the SQL needs to alias the columns differently for some reason. Note that in the `@FieldResult` the `name` is the name of the attribute in the object, not the column name in

---

20 <https://java.sun.com/javaee/5/docs/api/javax/persistence/SqlResultSetMapping.html>

21 <https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityResult.html>

22 <https://java.sun.com/javaee/5/docs/api/javax/persistence/ColumnResult.html>

23 <https://java.sun.com/javaee/5/docs/api/javax/persistence/FieldResult.html>

the mapping. This seems odd, because this would make mapping an **Embedded** or composite id relationship not possible.

Normally it is easiest to either select raw data or a single object with native SQL queries, so **@SqlResultSetMappings** can normally be avoided, as they are quite complex. Also note that even if you select the **Employee** and its **Address** with the SQL, these are two unrelated objects, the employee's address is not set, and may trigger a query if accessed unless a cache hit occurs. Some JPA providers may provide a query hint to allow join fetching to be used with native SQL queries.

### Example result set mapping annotation

```
@NamedNativeQuery(
    name="findAllEmployeesInCity",
    query="SELECT E.*, A.* from EMP E, ADDRESS A WHERE E.EMP_ID =
A.EMP_ID AND A.CITY = ?",
    resultSetMapping="employee-address"
)
@SqlResultSetMapping(name="employee-address",
    entities={
        @EntityResult(entityClass=Employee.class),
        @EntityResult(entityClass=Address.class)}
    )
public class Employee {
    ...
}
```

## 45.7 Stored Procedures

See Stored Procedures<sup>24</sup>

## 45.8 Raw JDBC

It can sometimes be required to mix JDBC code with JPA code. This may be to access certain JDBC driver specific features, or to integrate with another application that uses JDBC instead of JPA.

If you just require a JDBC connection, you could access one from your JEE server's **DataSource**, or connect directly to **DriverManager** or a third party connection pool. If you need a JDBC connection in the same transaction context and your JPA application, you could use a JTA **DataSource** for JPA and your JDBC access to have them share the same global transaction. If you are not using JEE, or not using JTA, then you may be able to access the JDBC connection directly from your JPA provider.

Some JPA providers provide an API to access a raw JDBC connection from their internal connection pool, or from their transaction context. In JPA 2.0 this API is somewhat standardized by the **unwrap** API on **EntityManager**.

---

<sup>24</sup> Chapter 41.4 on page 198

To access a JDBC connection from an `EntityManager`, some JPA 2.0 providers may support:

```
java.sql.Connection connection =  
    entityManager.unwrap(java.sql.Connection.class);
```

This connection could then be used for raw JDBC access. It normally should not be close when finished, as the connection is being used by the `EntityManager` and will be released when the `EntityManager` is closed or transaction committed.

Querying <sup>25</sup> Querying <sup>26</sup>

---

<sup>25</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

<sup>26</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FRuntime>

## 46 JPQL BNF

The following defines the structure of the JPQL query language. For further examples and usage see Querying<sup>1</sup>.

```
| = or  
[] = optional  
* = repeatable (zero or more times)  
{ } = mandatory
```

QL\_statement ::= select\_statement<sup>2</sup> | update\_statement<sup>3</sup> | delete\_statement<sup>4</sup>

### 46.1 Select

```
Select employee from Employee employee join employee.address address  
where address.city = :city and employee.firstName like :name order by  
employee.firstName
```

select\_statement ::= select\_clause<sup>5</sup> from\_clause<sup>6</sup> [where\_clause<sup>7</sup>] [groupby\_clause<sup>8</sup>]  
[having\_clause<sup>9</sup>] [orderby\_clause<sup>10</sup>]

#### 46.1.1 From

```
FROM Employee employee JOIN FETCH employee.address  
LEFT OUTER JOIN FETCH employee.phones JOIN employee.manager manager,  
Employee ceo
```

```
from_clause ::= FROM identification_variable_declaration {,  
{identification_variable_declaration |  
collection_member_declaration}}*
```

```
identification_variable_declaration ::= range_variable_declaration {  
join | fetch_join }*
```

- 
- 1 Chapter 43.2.3 on page 213
  - 2 Chapter 46.1.2 on page 234
  - 3 Chapter 46.2 on page 237
  - 4 Chapter 46.3 on page 238
  - 5 Chapter 46.1.2 on page 234
  - 6 Chapter 46.1.1 on page 233
  - 7 Chapter 46.1.3 on page 235
  - 8 Chapter 46.1.5 on page 237
  - 9 Chapter 46.1.5 on page 237
  - 10 Chapter 46.1.6 on page 237



```
range_variable_declaration ::= abstract_schema_name [AS]
                             identification_variable

join ::= join_spec join_association_path_expression [AS]
       identification_variable

fetch_join ::= join_spec FETCH join_association_path_expression

association_path_expression ::=
collection_valued_path_expression |
single_valued_association_path_expression

join_spec ::= [ LEFT [OUTER] | INNER ] JOIN

join_association_path_expression ::=
join_collection_valued_path_expression |
join_single_valued_association_path_expression

join_collection_valued_path_expression ::=
identification_variable.collection_valued_association_field

join_single_valued_association_path_expression ::=
identification_variable.single_valued_association_field

collection_member_declaration ::=
IN (collection_valued_path_expression) [AS] identification_variable

single_valued_path_expression ::=
state_field_path_expression |
single_valued_association_path_expression

state_field_path_expression ::=
{identification_variable |
single_valued_association_path_expression}.state_field

single_valued_association_path_expression ::=
identification_variable.{single_valued_association_field.}*
single_valued_association_field

collection_valued_path_expression ::=
identification_variable.{
single_valued_association_field.}*collection_valued_association_field

state_field ::= {embedded_class_state_field.}*simple_state_field
```

### 46.1.2 Select Clause

```
SELECT employee.id, employee.phones

SELECT DISTINCT employee.address.city, NEW
com.acme.EmployeeInfo(AVG(employee.salary), MAX(employee.salary))

select_clause ::= SELECT [DISTINCT] select_expression {, select_expression}*

select_expression ::= single_valued_path_expression | aggregate_expression |
identification_variable | OBJECT(identification_variable) | constructor_expression

constructor_expression ::= NEW constructor_name ( constructor_item {, constructor_
item}* )

constructor_item ::= single_valued_path_expression | aggregate_expression
```

aggregate\_expression ::= { AVG | MAX | MIN | SUM } ([DISTINCT] state\_field\_path\_expression) | COUNT ([DISTINCT] identification\_variable | state\_field\_path\_expression | single\_valued\_association\_path\_expression)

### 46.1.3 Where

```
WHERE employee.firstName = :name AND employee.address.city LIKE
    ,0tt%, ESCAPE '/',
OR employee.id IN (1, 2, 3) AND (employee.salary * 2) > 40000
```

where\_clause ::= WHERE conditional\_expression

conditional\_expression ::= conditional\_term | conditional\_expression OR conditional\_term

conditional\_term ::= conditional\_factor | conditional\_term AND conditional\_factor

conditional\_factor ::= [ NOT ] conditional\_primary

conditional\_primary ::= simple\_cond\_expression | (conditional\_expression)

simple\_cond\_expression ::= comparison\_expression | between\_expression | like\_expression | in\_expression | null\_comparison\_expression | empty\_collection\_comparison\_expression | collection\_member\_expression | exists\_expression

between\_expression ::= arithmetic\_expression [NOT] BETWEEN arithmetic\_expression AND arithmetic\_expression | string\_expression [NOT] BETWEEN string\_expression AND string\_expression | datetime\_expression [NOT] BETWEEN datetime\_expression AND datetime\_expression

in\_expression ::= state\_field\_path\_expression [NOT] IN ( in\_item {, in\_item}\* | subquery)

in\_item ::= literal | input\_parameter

like\_expression ::= string\_expression [NOT] LIKE pattern\_value [ESCAPE escape\_character]

null\_comparison\_expression ::= {single\_valued\_path\_expression | input\_parameter} IS [NOT] NULL

empty\_collection\_comparison\_expression ::= collection\_valued\_path\_expression IS [NOT] EMPTY

collection\_member\_expression ::= entity\_expression [NOT] MEMBER [OF] collection\_valued\_path\_expression

exists\_expression ::= [NOT] EXISTS (subquery)

all\_or\_any\_expression ::= { ALL | ANY | SOME } (subquery)

comparison\_expression ::= string\_expression comparison\_operator {string\_expression | all\_or\_any\_expression} | boolean\_expression { =|<> } {boolean\_expression | all\_or\_any\_expression} | enum\_expression { =|<> } {enum\_expression | all\_or\_any\_expression} | datetime\_expression comparison\_operator {datetime\_expression | all\_or\_any\_expression} | entity\_expression { =|<> } {entity\_expression | all\_or\_any\_expression} | arithmetic\_expression comparison\_operator {arithmetic\_expression | all\_or\_any\_expression}

comparison\_operator ::= = | > | >= | < | <= | <>

arithmetic\_expression ::= simple\_arithmetic\_expression | (subquery)

simple\_arithmetic\_expression ::= arithmetic\_term | simple\_arithmetic\_expression { + | - } arithmetic\_term

arithmetic\_term ::= arithmetic\_factor | arithmetic\_term { \* | / } arithmetic\_factor

arithmetic\_factor ::= [{ + | - }] arithmetic\_primary

arithmetic\_primary ::= state\_field\_path\_expression | numeric\_literal | (simple\_arithmetic\_expression) | input\_parameter | functions\_returning\_numerics | aggregate\_expression

string\_expression ::= string\_primary | (subquery)

string\_primary ::= state\_field\_path\_expression | string\_literal | input\_parameter | functions\_returning\_strings | aggregate\_expression

datetime\_expression ::= datetime\_primary | (subquery)

datetime\_primary ::= state\_field\_path\_expression | input\_parameter | functions\_returning\_datetime | aggregate\_expression

boolean\_expression ::= boolean\_primary | (subquery)

boolean\_primary ::= state\_field\_path\_expression | boolean\_literal | input\_parameter |

enum\_expression ::= enum\_primary | (subquery)

enum\_primary ::= state\_field\_path\_expression | enum\_literal | input\_parameter |

entity\_expression ::= single\_valued\_association\_path\_expression | simple\_entity\_expression

simple\_entity\_expression ::= identification\_variable | input\_parameter

#### 46.1.4 Functions

LENGTH(SUBSTRING(UPPER(CONCAT(,FOO,, :bar)), 1, 5))

functions\_returning\_numerics ::= LENGTH(string\_primary) | LOCATE(string\_primary, string\_primary[, simple\_arithmetic\_expression]) | ABS(simple\_arithmetic\_expression) | SQRT(simple\_arithmetic\_expression) | MOD(simple\_arithmetic\_expression, simple\_arithmetic\_expression) | SIZE(collection\_valued\_path\_expression)

functions\_returning\_datetime ::= CURRENT\_DATE | CURRENT\_TIME | CURRENT\_TIMESTAMP

functions\_returning\_strings ::= CONCAT(string\_primary, string\_primary) | SUBSTRING(string\_primary, simple\_arithmetic\_expression, simple\_arithmetic\_expression) | TRIM([[trim\_specification] [trim\_character] FROM] string\_primary) | LOWER(string\_primary) | UPPER(string\_primary)

trim\_specification ::= LEADING | TRAILING | BOTH

### 46.1.5 Group By

```
GROUP BY employee.address.country, employee.address.city HAVING
COUNT(employee.id) > 500
```

groupby\_clause ::= GROUP BY groupby\_item {, groupby\_item}\*

groupby\_item ::= single\_valued\_path\_expression | identification\_variable

having\_clause ::= HAVING conditional\_expression

### 46.1.6 Order By

```
ORDER BY employee.address.country, employee.address.city DESC
```

orderby\_clause ::= ORDER BY orderby\_item {, orderby\_item}\*

orderby\_item ::= state\_field\_path\_expression [ ASC | DESC ]

### 46.1.7 Subquery

```
WHERE employee.salary = (SELECT MAX(wellPaid.salary) FROM Employee
wellPaid)
```

subquery ::= simple\_select\_clause subquery\_from\_clause [where\_clause] [groupby\_clause] [having\_clause]

subquery\_from\_clause ::= FROM subselect\_identification\_variable\_declaration {, subselect\_identification\_variable\_declaration}\* subselect\_identification\_variable\_declaration ::= identification\_variable\_declaration | association\_path\_expression [AS] identification\_variable | collection\_member\_declaration

simple\_select\_clause ::= SELECT [DISTINCT] simple\_select\_expression

simple\_select\_expression ::= single\_valued\_path\_expression | aggregate\_expression | identification\_variable

## 46.2 Update

```
UPDATE Employee e SET e.salary = e.salary * 2 WHERE e.address.city =
:city
```

update\_statement ::= update\_clause [where\_clause<sup>11</sup>]

update\_clause ::= UPDATE abstract\_schema\_name [[AS] identification\_variable] SET update\_item {, update\_item}\*

---

<sup>11</sup> Chapter 46.1.3 on page 235

`update_item ::= [identification_variable.]{state_field | single_valued_association_field} = new_value`

`new_value ::= simple_arithmetic_expression | string_primary | datetime_primary | boolean_primary | enum_primary | simple_entity_expression | NULL`

## 46.3 Delete

```
DELETE FROM Employee e WHERE e.address.city = :city
```

`delete_statement ::= delete_clause [where_clause12]`

`delete_clause ::= DELETE FROM abstract_schema_name [[AS] identification_variable]`

## 46.4 Literals

It is normally best to define data values in a query using parameters, using the syntax `:parameter` or `?`. JPQL also allows for data values to be in-lined in the JPQL query using literals.

JPQL defines the following syntax for literals:

- String - 'string' - "Select e from Employee e where e.name = 'Bob'"
  - To define a ' (quote) character in a string, the quote is double quoted, i.e. 'Baie-D''Urfé'.
- Integer - digits - "Select e from Employee e where e.id = 1234"
- Long - +|-digitsL - "Select e from Employee e where e.id = 1234L"
- Float - +|-digits.decimaleexponentF - "Select s from Stat s where s.ratio > 3.14F"
- Double - +|-digits.decimaleexponentD - "Select s from Stat s where s.ratio > 3.14e32D"
- Boolean - TRUE | FALSE - "Select e from Employee e where e.active = TRUE"
- Date - {d'yyyy-mm-dd'} - "Select e from Employee e where e.startDate = {d'2012-01-03'}"
- Time - {t'hh:mm:ss'} - "Select e from Employee e where e.startTime = {t'09:00:00'}"
- Timestamp - {ts'yyy-mm-dd hh:mm:ss.nnnnnnnnn'} - "Select e from Employee e where e.version = {ts'2012-01-03 09:00:00.000000001'}"
- Enum - package.class.enum - "Select e from Employee e where e.gender = org.acme.Gender.MALE"
- null - NULL - "Update Employee e set e.manager = NULL where e.manager = :manager"

---

<sup>12</sup> Chapter 46.1.3 on page 235

## 46.5 New in JPA 2.0

The following new syntax was added in JPA 2.0.

`type_discriminator ::= TYPE(identification_variable | single_valued_object_path_expression | input_parameter)`

Allows type/class of object to be queried.

```
SELECT p from Project p where TYPE(p) in (LargeProject, SmallProject)
```

`qualified_identification_variable ::= KEY(identification_variable) | VALUE(identification_variable) | ENTRY(identification_variable)`

Allows selecting on Map keys, values and Map Entry.

```
SELECT ENTRY(e.contactInfo) from Employee e
```

`general_identification_variable ::= identification_variable | KEY(identification_variable) | VALUE(identification_variable)`

Allows querying on Map keys and values.

```
SELECT e from Employee e join e.contactInfo c where KEY(c) = ,Email,
and VALUE(c) = ,joe@gmail.com,
```

`in_item ::= literal | single_valued_input_parameter`

Allows collection parameters for IN.

```
SELECT e from Employee e where e.id in :param
```

`functions_returning_strings ::= CONCAT(string_primary, string_primary {, string_primary}*)`

Allows CONCAT with multiple arguments.

```
SELECT e from Employee e where CONCAT(e.address.street,
e.address.city, e.address.province) = :address
```

`SUBSTRING(string_primary, simple_arithmetic_expression [, simple_arithmetic_expression])`

Allows SUBSTRING with single argument.

```
SELECT e from Employee e where SUBSTRING(e.name, 3) = ,Mac,
```

`case_expression ::= general_case_expression | simple_case_expression | coalesce_expression | nullif_expression`  
`general_case_expression ::= CASE when_clause {when_clause}* ELSE scalar_expression END`  
`when_clause ::= WHEN conditional_expression THEN scalar_expression`  
`simple_case_expression ::= CASE case_operand simple_when_clause {simple_when_clause}* ELSE scalar_expression END`  
`case_operand ::= state_field_path_expression | type_discriminator`  
`simple_when_clause ::= WHEN scalar_expression THEN scalar_expression`  
`coalesce_expression ::= COALESCE(scalar_expression {, scalar_expression}+)`  
`nullif_expression ::= NULLIF(scalar_expression, scalar_expression)`

Allows CASE, COALESCE and NULLIF functions.

```
SELECT e.name, CASE WHEN (e.salary >= 100000) THEN 1 WHEN (e.salary <
100000) THEN 2 ELSE 0 END from Employee e
```

functions\_returning\_numerics ::= ... | INDEX(identification\_variable)

Allows querying in indexed List mapping's index.

```
SELECT e from Employee e join e.phones p where INDEX(p) = 1 and
p.areaCode = ,613,
```

join\_collection\_valued\_path\_expression ::= identification\_variable. {single\_valued\_ -  
embeddable\_object\_field.}\* collection\_valued\_field join\_single\_valued\_path\_ -  
expression ::= variable.{single\_valued\_embeddable\_object\_field.}\*single\_valued\_ -  
object\_field

Allows nested dot notation on joins.

```
SELECT p from Employee e join e.employeeDetails.phones p where e.id =
:id
```

select\_item ::= select\_expression [[AS] result\_variable]

Allows AS option in select.

```
SELECT AVG(e.salary) AS s, e.address.city from Employee e group by
e.address.city order by s
```

literalTemporal = DATE\_LITERAL | TIME\_LITERAL | TIMESTAMP\_LITERAL

Allows JDBC date/time escape syntax.

```
SELECT e from Employee e where e.startDate > {d,1990-01-01,}
```

JPQL BNF<sup>13</sup> JPQL BNF<sup>14</sup>

---

13 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

14 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FRuntime>

## 47 Persisting

JPA uses the `EntityManager`<sup>1</sup> API for runtime usage. The `EntityManager` represents the application session or dialog with the database. Each request, or each client will use its own `EntityManager` to access the database. The `EntityManager` also represents a transaction context, and in a typical stateless model a new `EntityManager` is created for each transaction. In a stateful model, an `EntityManager` may match the lifecycle of a client's session.

The `EntityManager` provides an API for all required persistence operations. These include the following CRUD operations:

- `persist`<sup>2</sup> (INSERT)
- `merge`<sup>3</sup> (UPDATE)
- `remove`<sup>4</sup> (DELETE)
- `find`<sup>5</sup> (SELECT)

The `EntityManager` is an object-oriented API, so does not map directly onto database SQL or DML operations. For example to update an object, you just need to read the object and change its state through its `set` methods, and then call `commit` on the transaction. The `EntityManager` figures out which objects you changed and performs the correct updates to the database, there is no explicit update operation in JPA.

### 47.1 Persist

The `EntityManager.persist()`<sup>6</sup> operation is used to insert a new object into the database. `persist` does not directly insert the object into the database, it just registers it as new in the persistence context (transaction). When the transaction is committed, or if the persistence context is *flushed*, then the object will be inserted into the database.

If the object uses a generated `Id`, the `Id` will normally be assigned to the object when `persist` is called, so `persist` can also be used to have an object's `Id` assigned. The one exception is if `IDENTITY` sequencing is used, in this case the `Id` is only assigned on `commit`

---

1 <https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html>  
2 [https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#persist\(java.lang.Object\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#persist(java.lang.Object))  
3 [https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#merge\(java.lang.Object\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#merge(java.lang.Object))  
4 [https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#remove\(java.lang.Object\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#remove(java.lang.Object))  
5 [https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#find\(java.lang.Class,%20java.lang.Object\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#find(java.lang.Class,%20java.lang.Object))  
6 [https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#persist\(java.lang.Object\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#persist(java.lang.Object))



or **flush** because the database will only assign the **Id** on **INSERT**. If the object does not use a generated **Id**, you should normally assign its **Id** before calling **persist**.

The **persist** operation can only be called within a transaction, an exception will be thrown outside of a transaction. The **persist** operation is in-place, in that the object being persisted will become part of the persistence context. The state of the object at the point of the commit of the transaction will be persisted, not its state at the point of the **persist** call.

**persist** should normally only be called on new objects. It is allowed to be called on existing objects if they are part of the persistence context, this is only for the purpose of cascading **persist** to any possible related new objects. If **persist** is called on an existing object that is not part of the persistence context, then an exception may be thrown, or it may be attempted to be inserted and a database constraint error may occur, or if no constraints are defined, it may be possible to have duplicate data inserted.

**persist** can only be called on **Entity** objects, not on **Embeddable** objects, or collections, or non-persistent objects. **Embeddable** objects are automatically persisted as part of their owning **Entity**.

Calling **persist** is not always required. If you related a new object to an existing object that is part of the persistence context, and the relationship is cascade **persist**, then it will be automatically inserted when the transaction is committed, or when the persistence context is flushed.

### Example persist

```
EntityManager em = getEntityManager();
em.getTransaction().begin();

Employee employee = new Employee();
employee.setFirstName("Bob");
Address address = new Address();
address.setCity("Ottawa");
employee.setAddress(address);

em.persist(employee);

em.getTransaction().commit();
```

#### 47.1.1 Cascading Persist

Calling **persist** on an object will also cascade the **persist** operation to across any relationship that is marked as cascade **persist**. If a relationship is not cascade **persist**, and a related object is new, then an exception may be thrown if you do not first call **persist** on the related object. Intuitively you may consider marking every relationship as cascade **persist** to avoid having to worry about calling **persist** on every objects, but this can also lead to issues.

One issue with marking all relationships cascade **persist** is performance. On each **persist** call all of the related objects will need to be traversed and checked if they reference any new objects. This can actually lead to  $n^2$  performance issues if you mark all relationships cascade **persist**, and **persist** a large new graph of objects. If you just call **persist** on the root object, this is ok. However, if you call **persist** on each object in the graph, then you

will traverse the entire graph for each object in the graph, and this can lead to a major performance issue. The JPA spec should probably define `persist` to only apply to new objects, not already part of the persistence context, but it requires `persist` apply to all objects, whether new, existing, or already persisted, so can have this issue.

A second issue is that if you `remove` an object to have it deleted, if you then call `persist` on the object, it will resurrect the object, and it will become persistent again. This may be desired if it is intentional, but the JPA spec also requires this behavior for cascade persist. So if you `remove` an object, but forget to remove a reference to it from a cascade persist relationship, the `remove` will be ignored.

I would recommend only marking relationships that are composite or privately owned as cascade persist.

## 47.2 Merge

The `EntityManager.merge()`<sup>7</sup> operation is used to merge the changes made to a detached object into the persistence context. `merge` does not directly update the object into the database, it merges the changes into the persistence context (transaction). When the transaction is committed, or if the persistence context is *flushed*, then the object will be updated in the database.

Normally `merge` is not required, although it is frequently misused. To update an object you simply need to read it, then change its state through its `set` methods, then commit the transaction. The `EntityManager` will figure out everything that has been changed and update the database. `merge` is only required when you have a detached copy of a persistence object. A *detached* object is one that was read through a different `EntityManager` (or in a different transaction in a JEE managed `EntityManager`), or one that was cloned, or serialized. A common case is a `stateless SessionBean` where the object is read in one transaction, then updated in another transaction. Since the update is processed in a different transaction, with a different `EntityManager`, it must first be merged. The `merge` operation will look-up/find the managed object for the detached object, and copy each of the detached objects attributes that changed into the managed object, as well as cascading any related objects marked as cascade merge.

The `merge` operation can only be called within a transaction, an exception will be thrown outside of a transaction. The `merge` operation is not in-place, in that the object being merged will never become part of the persistence context. Any further changes must be made to the managed object returned by the `merge`, not the detached object.

`merge` is normally called on existing objects, but can also be called on new objects. If the object is new, a new copy of the object will be made and registered with the persistence context, the detached object will not be persisted itself.

`merge` can only be called on `Entity` objects, not on `Embeddable` objects, or collections, or non-persistent objects. `Embeddable` objects are automatically merged as part of their owning `Entity`.

<sup>7</sup> [https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#merge\(java.lang.Object\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#merge(java.lang.Object))

## Example merge

```
EntityManager em = createEntityManager();
Employee detached = em.find(Employee.class, id);
em.close();
...
em = createEntityManager();
em.getTransaction().begin();
Employee managed = em.merge(detached);
em.getTransaction().commit();
```

### 47.2.1 Cascading Merge

Calling `merge` on an object will also cascade the `merge` operation across any relationship that is marked as cascade merge. Even if the relationship is not cascade merge, the reference will still be merged. If the relationship is cascade merge the relationship and each related object will be merged. Intuitively you may consider marking every relationship as cascade merge to avoid having to worry about calling merge on every objects, but this is normally a bad idea.

One issue with marking all relationships cascade merge is performance. If you have an object with a lot of relationships, then each `merge` call can require to traverse a large graph of objects.

Another issues arises if your detached object is corrupt in some way. For example say you have an `Employee` who has a `manager`, but that manager has a different copy of the detached `Employee` object as its `managedEmployee`. This may cause the same object to be merged twice, or at least may not be consistent which object will be merged, so you may not get the changes you expect merged. The same is true if you didn't change an object at all, but some other user did, if `merge` cascades to this unchanged object, it will revert the other user's changes, or throw an `OptimisticLockException` (depending on your locking policy). This is normally not desirable.

I would recommend only marking relationships that are composite or privately owned as cascade merge.

### 47.2.2 Transient Variables

Another issue with `merge` is transient variables. Since `merge` is normally used with object serialization, if a relationship was marked as `transient` (Java transient, not JPA transient), then the detached object will contain `null`, and `null` will be merged into the object, even though it is not desired. This will occur even if the relationship was not cascade merge, as `merge` always merges the references to related objects. Normally transient is required when using serialization to avoid serializing the entire database when only a single, or small set of objects are required.

One solution is to avoid marking anything `transient`, and instead use `LAZY` relationships in JPA to limit what is serialized (lazy relationships that have not been accessed, will normally not be serialized). Another solution is to manually merge in your own code.

Some JPA providers provide extended `merge` operations, such as allowing a *shallow* merge or *deep* merge, or merging without merging references.

## 47.3 Remove

The `EntityManager.remove()`<sup>8</sup> operation is used to delete an object from the database. `remove` does not directly delete the object from the database, it marks the object to be deleted in the persistence context (transaction). When the transaction is committed, or if the persistence context is *flushed*, then the object will be deleted from the database.

The `remove` operation can only be called within a transaction, an exception will be thrown outside of a transaction. The `remove` operation must be called on a managed object, not on a detached object. Generally you must first `find` the object before removing it, although it is possible to call `EntityManager.getReference()` on the object's `Id` and call `remove` on the reference. Depending on how you JPA provider optimizes `getReference` and `remove`, it may not require reading the object from the database.

`remove` can only be called on `Entity` objects, not on `Embeddable` objects, or collections, or non-persistent objects. `Embeddable` objects are automatically removed as part of their owning `Entity`.

### Example remove

```
EntityManager em = getEntityManager();
em.getTransaction().begin();
Employee employee = em.find(Employee.class, id);
em.remove(employee);
em.getTransaction().commit();
```

#### 47.3.1 Cascading Remove

Calling `remove` on an object will also cascade the `remove` operation across any relationship that is marked as cascade remove.

Note that cascade remove only effects the `remove` call. If you have a relationship that is cascade remove, and remove an object from the collection, or dereference an object, it will *not* be removed. You must explicitly call `remove` on the object to have it deleted. Some JPA providers provide an extension to provide this behavior, and in JPA 2.0 there will be an `orphanRemoval` option on `OneToMany` and `OneToOne` mappings to provide this.

#### 47.3.2 Reincarnation

Normally an object that has been removed, stays removed, but in some cases you may need to bring the object back to life. This normally occurs with natural ids, not generated ones,

<sup>8</sup> [https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#remove\(java.lang.Object\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#remove(java.lang.Object))

where a new object would always get an new id. Generally the desire to reincarnate an object occurs from a bad object model design, normally the desire to change the class type of an object (which cannot be done in Java, so a new object must be created). Normally the best solution is to change your object model to have your object hold a *type* object which defines its type, instead of using inheritance. But sometimes reincarnation is desirable.

When done in two separate transactions, this is normally fine, first you **remove** the object, then you **persist** it back. This can be more complex if you wish to **remove** and **persist** an object with the same **Id** in the same transaction. If you call **remove** on an object, then call **persist** on the same object, it will simply no longer be removed. If you call **remove** on an object, then call **persist** on a different object with the same **Id** the behavior may depend on your JPA provider, and probably will not work. If you call **flush** after calling **remove**, then call **persist**, then the object should be successfully reincarnated. Note that it will be a different row, the existing row will have been deleted, and a new row inserted. If you wish the same row to be updated, you may need to resort to using a native SQL update query.

## 48 Advanced

### 48.1 Refresh

The `EntityManager.refresh()`<sup>1</sup> operation is used to refresh an object's state from the database. This will revert any non-flushed changes made in the current transaction to the object, and refresh its state to what is currently defined on the database. If a **flush** has occurred, it will refresh to what was flushed. Refresh must be called on a managed object, so you may first need to **find** the object with the active `EntityManager` if you have a non-managed instance.

Refresh will cascade to any relationships marked **cascade** refresh, although it may be done lazily depending on your fetch type, so you may need to access the relationship to trigger the refresh. **refresh** can only be called on `Entity` objects, not on `Embeddable` objects, or collections, or non-persistent objects. `Embeddable` objects are automatically refreshed as part of their owning `Entity`.

Refresh can be used to revert changes, or if your JPA provider supports caching, it can be used to refresh stale cached data. Sometimes it is desirable to have a **Query** or **find** operation refresh the results. Unfortunately JPA 1.0 does not define how this can be done. Some JPA providers offer query hints to allow refreshing to be enabled on a query.

TopLink<sup>2</sup> / EclipseLink<sup>3</sup> : Define a query hint "`eclipselink.refresh`" to allow refreshing to be enabled on a query.

JPA 2.0 defines a set of standard query hints for refreshing, see JPA 2.0 Cache APIs<sup>4</sup>.

#### Example refresh

```
EntityManager em = getEntityManager();
em.refresh(employee);
```

### 48.2 Lock

See, Read and Write Locking<sup>5</sup>.

---

1 [https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#refresh\(java.lang.Object\)](https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#refresh(java.lang.Object))  
2 Chapter 11 on page 25  
3 Chapter 10 on page 23  
4 Chapter 51.6.4 on page 268  
5 Chapter 27.2.5 on page 106

## 48.3 Get Reference

The `EntityManager.getReference()`<sup>6</sup> operation is used to obtain a handle to an object without requiring it to be loaded. It is similar to the `find` operation, but may return a *proxy* or *unfetched* object. JPA does not require that `getReference` avoid loading the object, so some JPA providers may not support it and just perform a normal find operation. The object returned by `getReference` should appear to be a normal object, if you access any method or attribute other than its `Id` it will trigger itself to be refreshed from the database.

The intention of `getReference` is that it could be used on an insert or update operation as a stand-in for a related object, if you only have its `Id` and want to avoid loading the object. Note that `getReference` does not verify the existence of the object as `find` does. If the object does not exist and you try to use the *unfetched* object in an insert or update you may get a foreign key constraint violation, or if you access the object it may trigger an exception.

### Example `getReference`

```
EntityManager em = getEntityManager();
Employee manager = em.getReference(managerId);
Employee employee = new Employee();
...
em.persist(employee);
employee.setManager(manager);
em.commit();
```

## 48.4 Flush

The `EntityManager.flush()`<sup>7</sup> operation can be used to write all changes to the database before the transaction is committed. By default JPA does not normally write changes to the database until the transaction is committed. This is normally desirable as it avoids database access, resources and locks until required. It also allows database writes to be ordered, and batched for optimal database access, and to maintain integrity constraints and avoid deadlocks. This means that when you call `persist`, `merge`, or `remove` the database DML INSERT, UPDATE, DELETE is not executed, until commit, or until a flush is triggered.

Flush has several usages:

- Flush changes before a query execution to enable the query to return new objects and changes made in the persistence unit.
- Insert persisted objects to ensure their `Ids` are assigned and accessible to the application if using `IDENTITY` sequencing.
- Write all changes to the database to allow error handling of any database errors (useful when using JTA or `SessionBeans`).
- To flush and clear a batch for batch processing in a single transaction.
- Avoid constraint errors, or reincarnate an object.

---

<sup>6</sup> [https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#getReference\(java.lang.Class,%20java.lang.Object\)](https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#getReference(java.lang.Class,%20java.lang.Object))

<sup>7</sup> [https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#flush\(\)](https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#flush())

## Example flush

```
public long createOrder(Order order) throws ACMEException {
    EntityManager em = getEntityManager();
    em.persist(order);
    try {
        em.flush();
    } catch (PersistenceException exception) {
        throw new ACMEException(exception);
    }
    return order.getId();
}
```

## 48.5 Clear

The `EntityManager.clear()`<sup>8</sup> operation can be used to clear the persistence context. This will clear all objects read, changed, persisted, or removed from the current **EntityManager** or transaction. Changes that have already been written to the database through `flush`, or any changes made to the database will not be cleared. Any object that was read or persisted through the **EntityManager** is detached, meaning any changes made to it will not be tracked, and it should no longer be used unless merged into the new persistence context.

`clear` can be used similar to a *rollback* to abandon changes and restart a persistence context. If a transaction commit fails, or a rollback is performed the persistence context will automatically be cleared.

`clear` is similar to closing the **EntityManager** and creating a new one, the main difference being that `clear` can be called while a transaction is in progress. `clear` can also be used to free the objects and memory consumed by the **EntityManager**. It is important to note that an **EntityManager** is responsible for tracking and managing all objects read within its persistence context. In an application managed **EntityManager** this includes every objects read since the **EntityManager** was created, including *every* transaction the **EntityManager** was used for. If a long lived **EntityManager** is used, this is an intrinsic memory leak, so calling `clear` or closing the **EntityManager** and creating a new one is an important application design consideration. For JTA managed **EntityManager**s the persistence context is automatically cleared across each JTA transaction boundary.

Clearing is also important on large batch jobs, even if they occur in a single transaction. The batch job can be slit into smaller batches within the same transaction and `clear` can be called in between each batch to avoid the persistence context from getting too big.

## Example clear

```
public void processAllOpenOrders() {
    EntityManager em = getEntityManager();
    List<Long> openOrderIds = em.createQuery("SELECT o.id from Order o
    where o.isOpen = true");
    em.getTransaction().begin();
}
```

<sup>8</sup> [https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#clear\(\)](https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#clear())



```
try {
    for (int batch = 0; batch < openOrderIds.size(); batch += 100) {
        for (int index = 0; index < 100 && (batch + index) <
openOrderIds.size(); index++) {
            Long id = openOrderIds.get(batch + index);
            Order order = em.find(Order.class, id);
            order.process(em);
        }
        em.flush();
        em.clear();
    }
    em.getTransaction().commit();
} catch (RuntimeException error) {
    if (em.getTransaction().isActive()) {
        em.getTransaction().rollback();
    }
}
}
```

## 48.6 Close

The `EntityManager.close()`<sup>9</sup> operation is used to release an application managed `EntityManager`'s resources. JEE JTA managed `EntityManagers` cannot be closed, as they are managed by the JTA transaction and JEE server.

The life-cycle of an `EntityManager` can last either a transaction, request, or a users session. Typically the life-cycle is per request, and the `EntityManager` is closed at the end of the request. The objects obtained from an `EntityManager` become detached when the `EntityManager` is closed, and any LAZY relationships may no longer be accessible if they were not accessed before the `EntityManager` was closed. Some JPA providers allow LAZY relationships to be accessed after close.

### Example close

```
public Order findOrder(long id) {
    EntityManager em = factory.createEntityManager();
    Order order = em.find(Order.class, id);
    order.getOrderLines().size();
    em.close();
    return order;
}
```

## 48.7 Get Delegate

The `EntityManager.getDelegate()`<sup>10</sup> operation is used to access the JPA provider's `EntityManager` implementation class in a JEE managed `EntityManager`. A JEE managed `EntityManager` will be wrapped by a *proxy* `EntityManager` by the JEE server that forwards requests to the `EntityManager` active for the current JTA transaction. If a JPA provider

---

9 [https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#close\(\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#close())

10 [https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#getDelegate\(\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#getDelegate())

specific API is desired the `getDelegate()` API allows the JPA implementation to be accessed to call the API.

In JEE a managed `EntityManager` will typically create a new `EntityManager` per JTA transaction. Also the behavior is somewhat undefined outside of a JTA transaction context. Outside a JTA transaction context, a JEE managed `EntityManager` may create a new `EntityManager` per method, so `getDelegate()` may return a temporary `EntityManager` or even `null`. Another way to access the JPA implementation is through the `EntityManagerFactory`, which is typically not wrapped with a *proxy*, but may be in some servers.

In JPA 2.0 the `getDelegate()` API has been replaced by the `unwrap()` API which is more generic.

### Example `getDelegate`

```
public void clearCache() {
    EntityManager em = getEntityManager();
    ((JpaEntityManager)em.getDelegate()).get
    ServerSession().getIdentityMapAccessor().initializeAllIdentityMaps();
}
```

## 48.8 Unwrap (JPA 2.0)

The `EntityManager.unwrap()`<sup>11</sup> operation is used to access the JPA provider's `EntityManager` implementation class in a JEE managed `EntityManager`. A JEE managed `EntityManager` will be wrapped by a *proxy* `EntityManager` by the JEE server that forwards requests to the `EntityManager` active for the current JTA transaction. If a JPA provider specific API is desired the `unwrap()` API allows the JPA implementation to be accessed to call the API.

In JEE a managed `EntityManager` will typically create a new `EntityManager` per JTA transaction. Also the behavior is somewhat undefined outside of a JTA transaction context. Outside a JTA transaction context, a JEE managed `EntityManager` may create a new `EntityManager` per method, so `getDelegate()` may return a temporary `EntityManager` or even `null`. Another way to access the JPA implementation is through the `EntityManagerFactory`, which is typically not wrapped with a *proxy*, but may be in some servers.

### Example `unwrap`

```
public void clearCache() {
    EntityManager em = getEntityManager();
    em.unwrap(JpaEntityManager.class).get
```

<sup>11</sup> [https://java.sun.com/javase/6/docs/api/javax/persistence/EntityManager.html#unwrap\(java.lang.Class\)](https://java.sun.com/javase/6/docs/api/javax/persistence/EntityManager.html#unwrap(java.lang.Class))

```
ServerSession().getIdentityMapAccessor().initializeAllIdentityMaps();  
}
```

Persisting<sup>12</sup> Persisting<sup>13</sup>

kk:Java Persistence<sup>14</sup>

---

<sup>12</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FRuntime>

<sup>13</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

<sup>14</sup> <http://kk.wikibooks.org/wiki/Java%20Persistence>

## 49 Transactions

A transaction<sup>1</sup> is a set of operations that either fail or succeed as a unit. Transactions are a fundamental part of persistence. A database transaction consists of a set of SQL DML<sup>2</sup> (Data Manipulation Language) operations that are committed or rolled back as a single unit. An object level transaction is one in which a set of changes made to a set of objects are committed to the database as a single unit.

JPA provides two mechanisms for transactions. When used in JEE<sup>3</sup> JPA provides integration with JTA<sup>4</sup> (Java Transaction API). JPA also provides its own `EntityTransaction` implementation for JSE<sup>5</sup> and for use in a non-managed mode in JEE. Transactions in JPA are always at the object level, this means that all changes made to all persistent objects in the persistence context are part of the transaction.

### 49.1 Resource Local Transactions

Resource local transactions are used in JSE, or in application managed (non-managed) mode in JEE. To use resource local transactions the `transaction-type` attribute in the `persistence.xml` is set to `RESOURCE_LOCAL`. If resource local transactions are used with a `DataSource`, the `<non-jta-datasource>` element should be used to reference a server `DataSource` that has been configured to not be JTA managed.

Local JPA transactions are defined through the `EntityTransaction`<sup>6</sup> class. It contains basic transaction API including `begin`, `commit` and `rollback`.

Technically in JPA the `EntityManager` is in a transaction from the point it is created. So `begin` is somewhat redundant. Until `begin` is called, certain operations such as `persist`, `merge`, `remove` cannot be called. Queries can still be performed, and objects that were queried can be changed, although this is somewhat unspecified what will happen to these changes in the JPA spec, normally they will be committed, however it is best to call `begin` before making any changes to your objects. Normally it is best to create a new `EntityManager` for each transaction to avoid have stale objects remaining in the persistence context, and to allow previously managed objects to garbage collect.

After a successful `commit` the `EntityManager` can continue to be used, and all of the managed objects remain managed. However it is normally best to `close` or `clear` the `EntityManager`

---

1 <http://en.wikipedia.org/wiki/Database%20transaction>  
2 <http://en.wikipedia.org/wiki/Data%20Manipulation%20Language>  
3 <http://en.wikipedia.org/wiki/Java%20Enterprise%20Edition>  
4 <http://en.wikipedia.org/wiki/Java%20Transaction%20API>  
5 <http://en.wikipedia.org/wiki/Java%20Standard%20Edition>  
6 <https://java.sun.com/javase/5/docs/api/javax/persistence/EntityTransaction.html>

to allow garbage collection and avoid stale data. If the commit fails, then the managed objects are considered detached, and the `EntityManager` is cleared. This means that commit failures cannot be caught and retried, if a failure occurs, the entire transaction must be performed again. The previously managed object may also be left in an inconsistent state, meaning some of the objects locking version may have been incremented. Commit will also fail if the transaction has been marked for rollback. This can occur either explicitly by calling `setRollbackOnly` or is required to be set if *any* query or find operation fails. This can be an issue, as some queries may fail, but may not be desired to cause the entire transaction to be rolled back.

The `rollback` operation will rollback the database transaction *only*. The managed objects in the persistence context will become detached and the `EntityManager` is cleared. This means any object previously read, should no longer be used, and is no longer part of the persistence context. The changes made to the objects will be left as is, the object changes will *not* be reverted.

#### 49.1.1 Example resource local transaction persistence.xml

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
persistence_1_0.xsd" version="1.0">
    <persistence-unit name="acme" transaction-type="RESOURCE_LOCAL">
        <non-jta-data-source>amce</non-jta-data-source>
    </persistence-unit>
</persistence>
```

#### 49.1.2 Example resource local transaction

```
EntityManager em = createEntityManager();
em.getTransaction().begin();
Employee employee = em.find(Employee.class, id);
employee.setSalary(employee.getSalary() + 1000);
em.getTransaction().commit();
em.close();
```

### 49.2 JTA Transactions

JTA transactions are used in JEE, in managed mode (EJB<sup>7</sup>). To use JTA transactions the `transaction-type` attribute in the `persistence.xml` is set to `JTA`. If JTA transactions are used with a `DataSource`, the `<jta-datasource>` element should be used to reference a server `DataSource` that has been configured to be JTA managed.

JTA transactions are defined through the JTA `UserTransaction`<sup>8</sup> class, or more likely implicitly defined through `SessionBean` usage/methods. In a `SessionBean` normally each `SessionBean` method invocation defines a JTA transaction. `UserTransaction` can be

---

7 Chapter 43.2 on page 211

8 <http://java.sun.com/javase/5/docs/api/javax/transaction/UserTransaction.html>

obtained through a JNDI lookup in most application servers, or from the `EJBContext` in EJB 2.0 style `SessionBeans`.

JTA transactions can be used in two modes in JEE. In JEE managed mode, such as an `EntityManager` injected into a `SessionBean`, the `EntityManager` reference, represents a new persistence context for each transaction. This means objects read in one transaction become detached after the end of the transaction, and should no longer be used, or need to be merged into the next transaction. In managed mode, you never create or close an `EntityManager`.

The second mode allows the `EntityManager` to be application managed, (normally obtained from an injected `EntityManagerFactory`, or directly from JPA Persistence). This allows the persistence context to survive transaction boundaries, and follow the normal `EntityManager` life-cycle similar to resource local. If the `EntityManager` is created in the context of an active JTA transaction, it will automatically be part of the JTA transaction and commit/rollback with the JTA transaction. Otherwise it must join a JTA transaction to commit/rollback using `EntityManager.joinTransaction()`<sup>9</sup>

### 49.2.1 Example JTA transaction persistence.xml

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    persistence_1_0.xsd" version="1.0">
  <persistence-unit name="acme" transaction-type="JTA">
    <jta-data-source>amce</jta-data-source>
  </persistence-unit>
</persistence>
```

### 49.2.2 Example JTA transaction

```
UserTransaction transaction = (UserTransaction)new
  InitialContext().lookup("java:comp/UserTransaction");
transaction.begin();
EntityManager em = getEntityManager();
Employee employee = em.find(Employee.class, id);
employee.setSalary(employee.getSalary() + 1000);
transaction.commit();
```

<sup>9</sup> [https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#joinTransaction\(\)](https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#joinTransaction())



## 50 Advanced

### 50.1 Join Transaction

The `EntityManager.joinTransaction()`<sup>1</sup> API allows an application managed `EntityManager` to join the active JTA transaction context. This allows an `EntityManager` to be created outside the JTA transaction scope and commit its changes as part of the current transaction. This is normally used with a `stateful SessionBean`, or with a JSP or Servlet where an `EXTENDED EntityManager` is used in a *stateful* architecture. A *stateful* architecture is one where the server stores information on a client connection until the client's session is over, it differs from a *stateless* architecture where nothing is stored on the server in between client requests (each request is processed on its own).

There are pros and cons with both *stateful* and *stateless* architectures. One of the advantages with using a *stateful* architecture and `EXTENDED EntityManager`, is that you do not have to worry about merging objects. You can read your objects in one request, let the client modify them, and then commit them as part of a new transaction. This is where `joinTransaction` would be used. One issue with this is that you normally want to create the `EntityManager` when there is no active JTA transaction, otherwise it will commit as part of that transaction. However, even if it does commit, you can still continue to use it and join a future transaction. You do have to avoid using transactional API such as `merge` or `remove` until you are ready to commit the transaction.

`joinTransaction` is only used with JTA managed `EntityManagers` (JTA transaction-type in `persistence.xml`). For `RESOURCE_LOCAL` `EntityManagers` you can just commit the JPA transaction whenever you desire.

#### 50.1.1 Example joinTransaction usage

```
EntityManager em = getEntityManagerFactory().createEntityManager();
Employee employee = em.find(Employee.class, id);
employee.setSalary(employee.getSalary() + 1000);

UserTransaction transaction = (UserTransaction)new
    InitialContext().lookup("java:comp/UserTransaction");
transaction.begin();
em.joinTransaction();
transaction.commit();
```

---

<sup>1</sup> [https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#joinTransaction\(\)](https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#joinTransaction())



## 50.2 Retrying Transactions, Handling Commit Failures

Sometimes it is desirable to handle persistence errors, and recover and retry transactions. This normally requires a lot of application knowledge to know what failed, what state the system is in, and how to fix it.

Unfortunately JPA does not provide a direct way of handling commit failures or error handling. When a transaction commit fails, the transaction is automatically rolled back, and the persistence context cleared, and all managed objects detached. Not only is there no way to handle a commit failure, but if any error occurs in an query before the commit, the transaction will be marked for rollback, so there is no real way to handle any error. This is because any query could potentially change the state of the database, so JPA does not know if the database is in an invalid or inconsistent state so must rollback the transaction. As well if the commit fails, the state of the objects registered in the persistence context may also be inconsistent (such as partially committed objects having their optimistic lock versions incremented), so the persistence context is cleared to avoid further errors.

Some JPA providers may provide extended API to allow handling commit failures, or handling errors on queries.

There are some methods to generically handle commit failures and other errors. Error handling in general is normally easier when using `RESOURCE_LOCAL` transactions, and not when using JTA transactions.

One method of error handling is to call `merge` for each managed object after the commit fails into a new `EntityManager`, then try to commit the new `EntityManager`. One issue may be that any ids that were assigned, or optimistic lock versions that were assigned or incremented may need to be reset. Also, if the original `EntityManager` was `EXTENDED`, any objects that were in use would still become detached, and need to be reset.

Another more involved method to error handling is to always work with a non-transactional `EntityManager`. When it's time to commit, a new `EntityManager` is created, the non-transactional objects are merged into it, and the new `EntityManager` is committed. If the commit fails, only the state of the new `EntityManager` may be inconsistent, the original `EntityManager` will be unaffected. This can allow the problem to be corrected, and the `EntityManager` re-merged into another new `EntityManager`. If the commit is successful any commit changes can be merged back into the original `EntityManager`, which can then continue to be used as normal. This solution requires a fair bit of overhead, so should only be used if error handling is really required, and the JPA provider provides no alternatives.

## 50.3 Nested Transactions

JPA and JTA do not support nested transactions.

A nested transaction is used to provide a transactional guarantee for a subset of operations performed within the scope of a larger transaction. Doing this allows you to commit and abort the subset of operations independently of the larger transaction.

The rules to the usage of a nested transaction are as follows:

While the nested (child) transaction is active, the parent transaction may not perform any operations other than to commit or abort, or to create more child transactions.

Committing a nested transaction has no effect on the state of the parent transaction. The parent transaction is still uncommitted. However, the parent transaction can now see any modifications made by the child transaction. Those modifications, of course, are still hidden to all other transactions until the parent also commits.

Likewise, aborting the nested transaction has no effect on the state of the parent transaction. The only result of the abort is that neither the parent nor any other transactions will see any of the database modifications performed under the protection of the nested transaction.

If the parent transaction commits or aborts while it has active children, the child transactions are resolved in the same way as the parent. That is, if the parent aborts, then the child transactions abort as well. If the parent commits, then whatever modifications have been performed by the child transactions are also committed.

The locks held by a nested transaction are not released when that transaction commits. Rather, they are now held by the parent transaction until such a time as that parent commits.

Any database modifications performed by the nested transaction are not visible outside of the larger encompassing transaction until such a time as that parent transaction is committed.

The depth of the nesting that you can achieve with nested transaction is limited only by memory.

## 50.4 Transaction Isolation

Transactions<sup>2</sup> Transactions<sup>3</sup>

---

<sup>2</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

<sup>3</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FRuntime>



# 51 Caching

Caching is the most important performance optimization technique. There are many things that can be cached in persistence, objects, data, database connections, database statements, query results, meta-data, relationships, to name a few. Caching in object persistence normally refers to the caching of objects or their data. Caching also influences object identity, that is that if you read an object, then read the same object again you should get the identical object back (same reference).

JPA 1.0 does not define a shared object cache, JPA providers can support a shared object cache or not, however most do. Caching in JPA is required with-in a transaction or within an extended persistence context to preserve object identity, but JPA does not require that caching be supported across transactions or persistence contexts.

JPA 2.0 defines the concept of a shared cache. The `@Cacheable`<sup>1</sup> annotation or `cacheable` XML attribute can be used to enable or disable caching on a class.

## 51.0.1 Example JPA 2.0 Cacheable annotation

```
@Entity
@Cacheable
public class Employee {
    ...
}
```

The `SharedCacheMode`<sup>2</sup> enum can also be set in the `<shared-cache-mode>` XML element in the `persistence.xml` to configure the default cache mode for the entire persistence unit.

## 51.0.2 Example JPA 2.0 SharedCacheMode XML

```
<persistence-unit name="ACME">
  <shared-cache-mode>NONE</shared-cache-mode>
</persistence-unit>
```

There are two types of caching. You can cache the objects themselves including all of their structure and relationships, or you can cache their database row data. Both provide a benefit, however just caching the row data is missing a huge part of the caching benefit as the retrieval of each relationship typically involves a database query, and the bulk of the cost of reading an object is spent in retrieving its relationships.

---

1 <https://java.sun.com/javaee/6/docs/api/javax/persistence/Cacheable.html>

2 <http://download.oracle.com/javaee/6/api/javax/persistence/SharedCacheMode.html>

## 51.1 Object Identity

Object identity in Java means if two variables ( $x$ ,  $y$ ) refer to the same logical object, then `x == y` returns `true`. Meaning that both reference the same thing (both a pointer to the same memory location).

In JPA object identity is maintained within a transaction, and (normally) within the same **EntityManager**. The exception is in a JEE managed **EntityManager**, object identity is only maintained inside of a transaction.

So the following is true in JPA:

```
Employee employee1 = entityManager.find(Employee.class, 123);
Employee employee2 = entityManager.find(Employee.class, 123);
assert (employee1 == employee2);
```

This holds true no matter how the object is accessed:

```
Employee employee1 = entityManager.find(Employee.class, 123);
Employee employee2 =
    employee1.getManagedEmployees().get(0).getManager();
assert (employee1 == employee2);
```

In JPA object identity is *not* maintained across **EntityManagers**. Each **EntityManager** maintains its own persistence context, and its own transactional state of its objects.

So the following is true in JPA:

```
EntityManager entityManager1 = factory.createEntityManager();
EntityManager entityManager2 = factory.createEntityManager();
Employee employee1 = entityManager1.find(Employee.class, 123);
Employee employee2 = entityManager2.find(Employee.class, 123);
assert (employee1 != employee2);
```

Object identity is normally a good thing, as it avoids having your application manage multiple copies of objects, and avoids the application changing one copy, but not the other. The reason different **EntityManagers** or transactions (in JEE) don't maintain object identity is that each transaction must isolate its changes from other users of the system. This is also normally a good thing, however it does require the application to be aware of copies, detached objects and merging.

Some JPA products may have a concept of *read-only* objects, in which object identity may be maintained across **EntityManagers** through a shared object cache.

## 51.2 Object Cache

An object cache is where the Java objects (entities) are themselves cached. The advantage of an object cache is that the data is cached in the same format that it is used in Java. Everything is stored at the object-level and no conversion is required when obtaining a cache hit. With JPA the **EntityManager** must still copy the objects to and from the cache, as it must maintain its transaction isolation, but that is all that is required. The objects do not need to be re-built, and the relationships are already available.

With an object cache, transient data may also be cached. This may occur automatically, or may require some effort. If transient data is not desired, you may also need to clear the data when the object gets cached.

Some JPA products allow *read-only* queries to access the object cache directly. Some products only allow object caching of read-only data. Obtaining a cache hit on read-only data is extremely efficient as the object does not need to be copied, other than the look-up, no work is required.

It is possible to create your own object cache for your read-only data by loading objects from JPA into your own object cache or `JCache` implementation. The main issue, which is always the main issue in caching in general, is how to handle updates and stale cached data, but if the data is read-only, this may not be an issue.

TopLink<sup>3</sup> / EclipseLink<sup>4</sup> : Support an object cache. The object cache is on by default, but can be globally or selectively enabled or configured per class. The persistence unit property `"eclipselink.cache.shared.default"` can be set to `"false"` to disable the cache. Read-only queries are supported through the `"eclipselink.read-only"` query hint, entities can also be marked to always be read-only using the `@ReadOnly` annotation.

## 51.3 Data Cache

A data cache, caches the object's data, not the objects themselves. The data is normally a representation of the object's database row. The advantage of a data cache is that it is easier to implement as you do not have to worry about relationships, object identity, or complex memory management. The disadvantage of a data cache is that it does not store the data as it is used in the application, and does not store relationships. This means that on a cache hit, the object must still be built from the data, and the relationships fetched from the database. Some products that support a data cache, also support a relationship cache<sup>5</sup>, or query cache<sup>6</sup> to allow caching of relationships.

Hibernate<sup>7</sup> : Supports integration with a third party data cache. Caching is not enabled by default and a third party caching product such as Ehcache must be used to enable caching.

### 51.3.1 Caching Relationships

Some products support a separate cache for caching relationships. This is normally required for `OneToMany` and `ManyToMany` relationships. `OneToOne` and `ManyToOne` relationships normally do not need to be cached, as they reference the object's `Id`. However an inverse `OneToOne` will require the relationship to be cached, as it references the foreign key, not primary key.

---

3 Chapter 11 on page 25  
4 Chapter 10 on page 23  
5 Chapter 51.3.1 on page 263  
6 Chapter 51.5 on page 265  
7 Chapter 12 on page 27

For a relationship cache, the results normally only store the related object's Id, not the object, or its data (to avoid duplicate and stale data). The key of the relationship cache is the source object's Id and the relationship name. Sometimes the relationship is cached as part of the data cache, if the data cache stores a structure instead of a database row. When a cache hit occurs on a relationship, the related objects are looked up in the data cache one by one. A potential issue with this, is that if the related object is not in the data cache, it will need to be selected from the database. This could result in very poor database performance as the objects can be loaded one by one. Some product that support caching relationships also support batching the selects to attempt to alleviate this issue.

## 51.4 Cache Types

There are many different caching types. The most common is a LRU cache, or one that ejects the Least Recently Used objects and maintains a fixed size number of MRU (Most Recently Used) objects.

Some cache types include:

- LRU - Keeps X number of recently used objects in the cache.
- Full - Caches everything read, forever. (not always the best idea if the database is large)
- Soft - Uses Java garbage collection hints to release objects from the cache when memory is low.
- Weak - Normally relevant with object caches, keeps any objects currently in use in the cache.
- L1<sup>8</sup> - This refers to the transactional cache that is part of every `EntityManager`, this is not a shared cache.
- L2<sup>9</sup> - This is a shared cache, conceptually stored in the `EntityManagerFactory`, so accessible to all `EntityManager`s.
- Data cache<sup>10</sup> - The data representing the objects is cached (database rows).
- Object cache<sup>11</sup> - The objects are cached directly.
- Relationship cache<sup>12</sup> - The object's relationships are cached.
- Query cache<sup>13</sup> - The result set from a query is cached.
- Read-only - A cache that only stores, or only allows read-only objects.
- Read-write - A cache that can handle insert, updates and deletes (non read-only).
- Transactional - A cache that can handle insert, updates and deletes (non read-only), and obeys transactional ACID properties.
- Clustered<sup>14</sup> - Typically refers to a cache that uses JMS, JGroups or some other mechanism to broadcast invalidation messages to other servers in the cluster when an object is updated or deleted.

---

8 Chapter 51.6.1 on page 266

9 Chapter 51.6.2 on page 266

10 Chapter 51.3 on page 263

11 Chapter 51.2 on page 262

12 Chapter 51.3.1 on page 263

13 Chapter 51.5 on page 265

14 Chapter 51.6.6 on page 269

- Replicated - Typically refers to a cache that uses JMS, JGroups or some other mechanism to broadcast objects to all servers when read into any of the servers cache.
- Distributed<sup>15</sup> - Typically refers to a cache that spreads the cached objects across several servers in a cluster, and can look-up an object in another server's cache.

TopLink<sup>16</sup> / EclipseLink<sup>17</sup> : Support an L1 and L2 object cache. LRU, Soft, Full and Weak cache types are supported. A query cache is supported. The object cache is read-write and always transactional. Support for cache coordination through RMI and JMS is provided for clustering. The TopLink<sup>18</sup> product includes a *Grid* component that integrates with Oracle Coherence to provide a distributed cache.

## 51.5 Query Cache

A query cache caches query results instead of objects. Object caches cache the object by its `Id`, so are generally not very useful for queries that are not by `Id`. Some object caches support secondary indexes, but even indexed caches are not very useful for queries that can return multiple objects, as you always need to access the database to ensure you have all of the objects. This is where query caches are useful, instead of storing objects by `Id`, the query results are cached. The cache key is based on the query name and parameters. So if you have a `NamedQuery` that is commonly executed, you can cache its results, and only need to execute the query the first time.

The main issue with query caches, as with caching in general is stale data. Query caches normally interact with an object cache to ensure the objects are at least as up to date as in the object cache. Query caches also typically have invalidation options similar to object caches.

TopLink<sup>19</sup> / EclipseLink<sup>20</sup> : Support query cache enabled through the query hint `"eclipselink.query-results-cache"`. Several configuration options including invalidation are supported.

## 51.6 Stale Data

The main issue with caching anything is the possibility of the cache version getting out of synch with the original. This is referred to as *stale* or *out of synch* data. For *read-only data*, *this is not an issue, but for data that changes in-frequently or frequently this can be a major issue. There are many techniques for dealing with stale data and out of synch data.*

---

15 Chapter 51.6.6 on page 270

16 Chapter 11 on page 25

17 Chapter 10 on page 23

18 Chapter 11 on page 25

19 Chapter 11 on page 25

20 Chapter 10 on page 23



### 51.6.1 1st Level Cache

*Caching the object's state for the duration of a transaction or request is normally not an issue. This is normally called a 1st level cache, or the **EntityManager** cache, and is required by JPA for proper transaction semantics. If you read the same object twice, you must get the identical object, with the same in-memory changes. The only issues occur with querying and DML.*

*For queries that access the database, the query may not reflect the un-written state of the objects. For example you have persisted a new object, but JPA has not yet inserted this object in the database as it generally only writes to the database on the commit of the transaction. So your query will not return this new object, as it is querying the database, not the 1st level cache. This is normally solved in JPA by the user first calling `flush()`, or the `flushMode` automatically triggering a flush. The default `flushMode` on an **EntityManager** or **Query** is to trigger a flush, but this can be disabled if a write to the database before every query is not desired (normally it isn't, as it can be expensive and lead to poor concurrency). Some JPA providers also support conforming the database query results with the object changes in memory, which can be used to get consistent data without triggering a flush. This can work for simple queries, but for complex queries this typically gets very complex to impossible. Applications normally query data at the start of a request before they start making changes, or don't query for objects they have already found, so this is normally not an issue.*

If you bypass JPA and execute DML directly on the database, either through native SQL queries, JDBC, or JPQL `UPDATE` or `DELETE` queries, then the database can be out of synch with the 1st level cache. If you had accessed objects before executing the DML, they will have the old state and not include the changes. Depending on what you are doing this may be ok, otherwise you may want to refresh the affected objects from the database.

The 1st level, or **EntityManager** cache can also span transaction boundaries in JPA. A JTA managed **EntityManager** will only exist for the duration of the JTA transaction in JEE. Typically the JEE server will inject the application with a proxy to an **EntityManager**, and after each JTA transaction a new **EntityManager** will be created automatically or the **EntityManager** will be cleared, clearing the 1st level cache. In an application managed **EntityManager**, the 1st level cache will exist for the duration of the **EntityManager**. This can lead to stale data, or even memory leaks and poor performance if the **EntityManager** is held too long. This is why it is generally a good idea to create a new **EntityManager** per request, or per transaction. The 1st level cache can also be cleared using the **EntityManager.clear()** method, or an object can be refreshed using the **EntityManager.refresh()** method.

### 51.6.2 2nd Level Cache

The 2nd level cache spans transactions and **EntityManagers**, and is not required as part of JPA. Most JPA providers support a 2nd level cache, but the implementation and semantics vary. Some JPA providers default to enabling a 2nd level cache, and some do not use a 2nd level cache by default.

If the application is the only application and server accessing the database there is little issue with the 2nd level cache, as it should always be up to date. The only issue is with

DML, if the application executes DML directly to the database through native SQL queries, JDBC, or JPQL `UPDATE` or `DELETE` queries. JPQL queries should automatically invalidate the 2nd level cache, but this may depend on the JPA provider. If you use native DML queries or JDBC directly, you may need to invalidate, refresh or clear the objects affected by the DML.

If there are other applications, or other application servers accessing the same database, then stale data can become a bigger issue. Read-only objects, and inserting new objects should not be an issue. New objects should get picked up by other servers even when using caching as queries typically still access the database. It is normally only `find()` operations and relationships that hit the cache. Updated and deleted objects by other applications or servers can cause the 2nd level cache to become stale.

For deleted objects, the only issue is with `find()` operations, as queries that access the database will not return the deleted objects. A `find()` by the object's `Id` could return the object if it is cached, even if it does not exist. This could lead to constraint issues if you add relations to this object from other objects, or failed updates, if you try to update the object. Note that these can both occur without caching, even with a single application and server accessing the database. During a transaction, another user of the application could always delete the object being used by another transaction, and the second transaction will fail in the same way. The difference is the potential for this concurrency issue to occur increases.

For updated objects, any query for the objects can return stale data. This can trigger optimistic lock exceptions on updates, or cause one user to overwrite another user's changes if not using locking. Again note that these can both occur without caching, even with a single application and server accessing the database. This is why it is normally always important to use optimistic locking. Stale data could also be returned to the user.

### 51.6.3 Refreshing

Refreshing is the most common solution to stale data. Most application users are familiar with the concept of a cache, and know when they need fresh data and are willing to click a refresh button. This is very common in an Internet browser, most browsers have a cache of web pages that have been accessed, and will avoid loading the same page twice, unless the user clicks the refresh button. This same concept can be used in building JPA applications. JPA provides several refreshing options, see refreshing<sup>21</sup>.

Some JPA providers also support refreshing options in their 2nd level cache. One option is to always refresh on any query to the database. This means that `find()` operations will still access the cache, but if the query accesses the database and brings back data, the 2nd level cache will be refreshed with the data. This avoids queries returning stale data, but means there will be less benefit from caching. The cost is not just in refreshing the objects, but in refreshing their relationships. Some JPA providers support this option in combination with optimistic locking. If the version value in the row from the database is newer than the version value from the object in the cache, then the object is refreshed as it is stale, otherwise the cache value is returned. This option provides optimal caching, and avoids stale data on queries. However objects returned through `find()` or through relationships

---

21 Chapter 48.1 on page 247

can still be stale. Some JPA providers also allow `find()` operation to be configured to first check the database, but this general defeats the purpose of caching, so you are better off not using a 2nd level cache at all. If you want to use a 2nd level cache, then you must have some level of tolerance to stale data.

#### 51.6.4 JPA 2.0 Cache APIs

JPA 2.0 provides a set of standard query hints to allow refreshing or bypassing the cache. The query hints are defined on the two enum classes `CacheRetrieveMode`<sup>22</sup> and `CacheStoreMode`<sup>23</sup>.

Query hints:

- `javax.persistence.cache.retrieveMode : CacheRetrieveMode`
  - `BYPASS` : Ignore the cache, and build the object directly from the database result.
  - `USE` : Allow the query to use the cache. If the object/data is already in the cache, the cached object/data will be used.
- `javax.persistence.cache.storeMode : CacheStoreMode`
  - `BYPASS` : Do not cache the database results.
  - `REFRESH` : If the object/data is already in the cache, then refresh/replace it with the database results.
  - `USE` : Cache the objects/data returned from the query.

#### Cache hints example

```
Query query = em.createQuery("Select e from Employee e");
query.setHint("javax.persistence.cache.storeMode", "REFRESH");
```

JPA 2.0 also provides a `Cache`<sup>24</sup> interface. The `Cache` interface can be obtained from the `EntityManagerFactory` using the `getCache()` API. The `Cache` can be used to manually evict/invalidate entities in the cache. Either a specific entity, an entire class, or the entire cache can be evicted. The `Cache` can also be checked to see if it contains an entity.

Some JPA providers may extend the `getCache()` interface to provide additional API.

`TopLink`<sup>25</sup> / `EclipseLink`<sup>26</sup> : Provide an extended `Cache` interface `JpaCache`. It provides additional API for invalidation, query cache, access and clearing.

#### Cache evict example

```
Cache cache = factory.getCache();
cache.evict(Employee.class, id);
```

---

<sup>22</sup> <https://java.sun.com/javase/6/docs/api/javax/persistence/CacheRetrieveMode.html>

<sup>23</sup> <https://java.sun.com/javase/6/docs/api/javax/persistence/CacheStoreMode.html>

<sup>24</sup> <https://java.sun.com/javase/6/docs/api/javax/persistence/Cache.html>

<sup>25</sup> Chapter 11 on page 25

<sup>26</sup> Chapter 10 on page 23

### 51.6.5 Cache Invalidation

A common way to deal with stale cached data is to use cache invalidation. Cache invalidation removes or invalidates data or objects in the cache after a certain amount of time, or at a certain time of day. Time to live invalidation guarantees that the application will never read cached data that is older than a certain amount of time. The amount of the time can be configured with respect to the application's requirements. Time of day invalidation allows the cache to be invalidated at a certain time of day, typically done in the night, this ensures data is never older than a day old. This can also be used if it is known that a batch job updates the database a night, the invalidation time can be set after this batch job is scheduled to run. Data can also be invalidated manually, such as using the JPA 2.0 `evict()` API.

Most cache implementations support some form of invalidation, JPA does not define any configurable invalidation options, so this depends on the JPA and cache provider.

TopLink<sup>27</sup> / EclipseLink<sup>28</sup> : Provide support for time to live and time of day cache invalidation using the `@Cache` annotation and `<cache>` orm.xml element. Cache invalidation is also supported through API, and can be used in a cluster to invalidate objects changed on other machines.

### 51.6.6 Caching in a Cluster

Caching in a clustered environment is difficult because each machine will update the database directly, but not update the other machine's caches, so each machine's cache can become out of date. This does not mean that caching cannot be used in a cluster, but you must be careful in how it is configured.

For read-only objects caching can still be used. For read mostly objects, caching can be used, but some mechanism should be used to avoid stale data. If stale data is only an issue for writes, then using optimistic locking will avoid writes occurring on stale data. When an optimistic lock exception occurs, some JPA providers will automatically refresh or invalidate the object in the cache, so if the user or application retries the transaction the next write will succeed. Your application could also catch the lock exception and refresh or invalidate the object, and potentially retry the transaction if the user does not need to be notified of the lock error (be careful doing this though, as normally the user should be aware of the lock error). Cache invalidation can also be used to decrease the likelihood of stale data by setting a time to live on the cache. The size of the cache can also affect the occurrence of stale data.

Although returning stale data to a user may be an issue, normally returning stale data to a user that just updated the data is a bigger issue. This can normally be solved through session infinity, but ensuring the user interacts with the same machine in the cluster for the duration of their session. This can also improve cache usage, as the same user will typically access the same data. It is normally also useful to add a *refresh* button to the UI, this will allow the user to refresh their data if they think their data is stale, or they wish to ensure

---

<sup>27</sup> Chapter 11 on page 25

<sup>28</sup> Chapter 10 on page 23

they have data that is up to date. The application can also choose to refresh the objects in places where up to date data is important, such as using the cache for read-only queries, but refreshing when entering a transaction to update an object.

For write mostly objects, the best solution may be to disable the cache for those objects. Caching provides no benefit to inserts, and the cost of avoiding stale data on updates may mean there is no benefit to caching objects that are always updated. Caching will add some overhead to writes, as the cache must be updated, having a large cache also affects garbage collection, so if the cache is not providing any benefit it should be turned off to avoid this overhead. This can depend on the complexity of the object though, if the object has a lot of complex relationships, and only part of the object is updated, then caching may still be worth it.

### Cache Coordination

One solution to caching in a clustered environment is to use a messaging framework to coordinate the caches between the machines in the cluster. JMS or JGroups can be used in combination with JPA or application events to broadcast messages to invalidate the caches on other machines when an update occurs. Some JPA and cache providers support cache coordination in a clustered environment.

TopLink<sup>29</sup> / EclipseLink<sup>30</sup> : Support cache coordination in a clustered environment using JMS or RMI. Cache coordination is configured through the `@Cache` annotation or `<cache>` orm.xml element, and using the persistence unit property `eclipselink.cache.coordination.protocol`.

### Distributed Caching

A distributed cache is one where the cache is distributed across each of the machines in the cluster. Each object will only live on one or a set number of the machines. This avoids stale data, because when the cache is accessed or updated the object is always retrieved from the same location, so is always up to date. The draw back to this solution is that cache access now potentially requires a network access. This solution works best when the machines in the cluster are connected together on the same high speed network, and when the database machine is not as well connected, or is under load. A distributed cache reduces database access, so allows the application to be scaled to a larger cluster without the database becoming a bottleneck. Some distributed cache providers also provide a local cache, and offer cache coordination between the caches.

TopLink<sup>31</sup> : Supports integration with the Oracle Coherence distributed cache.

---

<sup>29</sup> Chapter 11 on page 25

<sup>30</sup> Chapter 10 on page 23

<sup>31</sup> Chapter 11 on page 25

## 51.7 Cache Transaction Isolation

When caching is used, the consistency and isolation of the cache becomes as important as the database transaction isolation. For basic cache isolation it is important that changes are not committed into the cache until after the database transaction has been committed, otherwise uncommitted data could be accessed by other users.

Caches can be either transactional, or non-transactional. In a transactional cache, the changes from a transaction are committed to the cache as a single atomic unit. This means the objects/data are first locked in the cache (preventing other threads/users from accessing the objects/data), then updated in the cache, then the locks are released. Ideally the locks are obtained before committing the database transaction, to ensure consistency with the database. In a non-transactional cache the objects/data are updated one by one without any locking. This means there will be a brief period where the data in the cache is not consistent with the database. This may or may not be an issue, and is a complex issue to think about and discuss, and gets into the issue of locking, and the application's isolation requirements.

Optimistic locking is another important consideration in cache isolation. If optimistic locking is used, the cache should avoid replacing new data, with older data. This is important when reading, and when updating the cache.

Some JPA providers may allow for configuration of their cache isolation, or different caches may define different levels of isolation. Although the defaults should normally be used, it can be important to understand how the usage of caching is affecting your transaction isolation, as well as your performance and concurrency.

## 51.8 Common Problems

### *I can't see changes made directly on the database, or from another server*

This means you have either enabled caching in your JPA configuration, or your JPA provider caches by default. You can either disable the 2nd level cache in your JPA configuration, or refresh the object or invalidate the cache after changing the database directly. See, Stale Data<sup>32</sup>

TopLink<sup>33</sup> / EclipseLink<sup>34</sup> : Caching is enabled by default. To disable caching set the persistence property "`eclipselink.cache.shared.default`" to `false` in your persistence.xml or persistence properties. You can also configure this on a per class basis if you want to allow caching in some classes and not in others. See, EclipseLink FAQ<sup>35</sup>.

Caching<sup>36</sup> Caching<sup>37</sup>

---

<sup>32</sup> Chapter 51.6 on page 265

<sup>33</sup> Chapter 11 on page 25

<sup>34</sup> Chapter 10 on page 23

<sup>35</sup> [http://wiki.eclipse.org/EclipseLink/FAQ/How\\_to\\_disable\\_the\\_shared\\_cache%3F](http://wiki.eclipse.org/EclipseLink/FAQ/How_to_disable_the_shared_cache%3F)

<sup>36</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

<sup>37</sup> <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence%2FRuntime>



## 52 Spring

Spring<sup>1</sup> is an application framework for Java. Spring is an IoC<sup>2</sup> container that allows for a different programming model. Spring is similar to a JEE server, in that it provides a transaction service, XML deployment, annotation processing, byte-code weaving, and JPA integration.

### 52.1 Persistence

Persistence in Spring is normally done through a DAO (Data Access Object) layer. The Spring DAO layer is meant to encapsulate the persistence mechanism, so the same application data access API would be given no matter if JDBC, JPA or a native API were used.

Spring also defines a transaction manager implementation that is similar to JTA. Spring also supports transactional annotations and beans similar to SessionBeans.

### 52.2 JPA

Spring has specific support for JPA and can emulate some of the functionality of a JEE container with respect to JPA. Spring allows a JPA persistence unit to be deployed in container managed mode. If the spring-agent is used to start the JVM, Spring can deploy a JPA persistence unit with weaving similar to a JEE server. Spring can also pass a Spring DataSource and integrate its transaction service with JPA. Spring allows the `@PersistenceUnit` and `@PersistenceContext` annotations to be used in any Spring bean class, to have an `EntityManager` or `EntityManagerFactory` injected. Spring supports a managed transactional `EntityManager` similar to JEE, where the `EntityManager` binds itself as a new persistence context to each new transaction and commits as part of the transaction. Spring supports both JTA integration, and its own transaction manager.

#### 52.2.1 Example Spring JPA Deployment

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
```

---

1 [http://en.wikipedia.org/wiki/Spring\\_Framework](http://en.wikipedia.org/wiki/Spring_Framework)

2 [http://en.wikipedia.org/wiki/Inversion\\_of\\_Control](http://en.wikipedia.org/wiki/Inversion_of_Control)



```
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean id="entityManagerFactory" class=
"org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="persistenceUnitName" value="acme" />
      <property name="persistenceUnitManager"
ref="persistenceUnitManager" />
    </bean>

    <bean id="persistenceUnitManager" class="org.sp
ringframework.orm.jpa.persistenceunit.DefaultPersistenceUnitManager">
      <property name="defaultDataSource" ref="dataSource" />
      <property name="dataSources">
        <map>
          <entry>
            <key>
              <value>jdbc/__default</value>
            </key>
            <ref bean="dataSource" />
          </entry>
          <entry>
            <key>
              <value>jdbc/jta</value>
            </key>
            <ref bean="dataSource" />
          </entry>
        </map>
      </property>
      <property name="loadTimeWeaver">
        <bean class="org.s
pringframework.instrument.classloading.InstrumentationLoadTimeWeaver"
/>
      </property>
    </bean>

    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name="driverClassName"
value="oracle.jdbc.OracleDriver" />
      <property name="url"
value="jdbc:oracle:thin:@localhost:1521:orcl" />
      <property name="username" value="scott" />
      <property name="password" value="tiger" />
    </bean>

    <bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory"
ref="entityManagerFactory" />
    </bean>

    <bean id="entityManager"
class="org.springframework.orm.jpa.support.SharedEntityManagerBean">
      <property name="entityManagerFactory"
ref="entityManagerFactory"/>
    </bean>

    <bean id="AcmeDao" class="org.acme.dao.AcmeDao">
      <property name="entityManager" ref="entityManager" />
    </bean>

</beans>
```

Spring<sup>3</sup>

---

3 <http://en.wikibooks.org/wiki/Category%3AJava%20Persistence>

## 53 Databases



## 54 MySQL

Using MySQL with Java Persistence API is quite straightforward as the JDBC driver is available directly from MySQL web site <http://dev.mysql.com/>.

MySQL Connector/J is the official JDBC driver for MySQL and has a good documentation available directly on the web site.

In this page we will see some aspects of managing MySQL using the Persistence API.

### 54.1 Installing

Installation is straightforward and consists in making the .jar file downloaded from MySQL web site visible to the JVM. It can be already installed if you are using Apache or JBOSS.

### 54.2 Configuration tips

You can learn a lot from the documentation on MySQL web site <http://dev.mysql.com/doc/refman/5.0/en/connector-j-reference-configuration-properties.html>.

#### 54.2.1 Creating the database automatically

If you intend to create table and a database automatically , you will need to have the correct user rights but also inform the Persistence API about the name of the Database you want to create. For example with TopLink, if you used:

```
<property name="toplink.ddl-generation" value="create-tables"/>
```

in the property.xml, you will be likely need to create a new database. To create the database "NewDB" automatically, you need to give the following URL to the jdbc connection:

```
<property name="toplink.jdbc.url" value="jdbc:mysql://localhost:3306/NewDB?createDatabaseIfNotExist=true"/>
```

If not, the persistence API will complain that the database does not exist.



## 55 References



### 55.0.2 Resources

\* EJB 3.0 JPA 1.0 Spec<sup>1</sup>\* JPA 2.0 Spec<sup>2</sup>\* JPA 1.0 ORM XML Schema<sup>3</sup>\* JPA 1.0 Persistence XML Schema<sup>4</sup>\* JPA 1.0 JavaDoc<sup>5</sup>\* JPQL BNF<sup>6</sup>\* JPA 2.0 Reference Implementation Development<sup>7</sup>\* Java Programming<sup>8</sup>



### 55.0.3 Wikis

\* EclipseLink Wiki<sup>9</sup>\* Oracle TopLink Wiki<sup>10</sup>\* Glassfish TopLink Essentials Wiki<sup>11</sup>\* Hibernate Wiki<sup>12</sup>\* JPA on Wikipedia<sup>13</sup>\* JPA on Javapedia<sup>14</sup>\* JPA on freebase<sup>15</sup>\* JPA on DMOZ (Open Directory)<sup>16</sup>

- 
- 1 <http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html>
  - 2 <http://jcp.org/en/jsr/detail?id=317>
  - 3 [http://java.sun.com/xml/ns/persistence/orm\\_1\\_0.xsd](http://java.sun.com/xml/ns/persistence/orm_1_0.xsd)
  - 4 [http://java.sun.com/xml/ns/persistence/persistence\\_1\\_0.xsd](http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd)
  - 5 <https://java.sun.com/javaee/5/docs/api/javax/persistence/package-summary.html>
  - 6 <http://en.wikibooks.org/wiki/%2FJPQL%20BNF>
  - 7 <http://wiki.eclipse.org/EclipseLink/Development/JPA>
  - 8 <http://en.wikibooks.org/wiki/Java%20Programming>
  - 9 <http://wiki.eclipse.org/EclipseLink>
  - 10 <http://wiki.oracle.com/page/TopLink>
  - 11 <http://wiki.glassfish.java.net/Wiki.jsp?page=TopLinkEssentials>
  - 12 <http://www.hibernate.org/37.html>
  - 13 [http://en.wikipedia.org/wiki/Java\\_Persistence\\_API](http://en.wikipedia.org/wiki/Java_Persistence_API)
  - 14 <http://wiki.java.net/bin/view/Javapedia/JPA>
  - 15 <http://www.freebase.com/view/guid/9202a8c04000641f8000000004666d33>
  - 16 [http://www.dmoz.org/Computers/Programming/Languages/Java/Databases\\_and\\_Persistence/Object\\_Persistence/JPA/](http://www.dmoz.org/Computers/Programming/Languages/Java/Databases_and_Persistence/Object_Persistence/JPA/)



#### 55.0.4 Forums

\* Sun EJB Forum<sup>17</sup>\* JavaRanch ORM Forum<sup>18</sup>\*  
Nabble JPA Forum<sup>19</sup>\* EclipseLink Forum<sup>20</sup>\*  
EclipseLink Newsgroup<sup>21</sup>\* Oracle TopLink Forum<sup>22</sup>\*  
Hibernate Forum<sup>23</sup>\* TopLink Essentials Mailing List  
(Glassfish persistence)<sup>24</sup>

#### 55.0.6 Blogs

\* Java Persistence<sup>31</sup> (Doug Clarke)\* System.out<sup>32</sup>  
(Mike Keith)\* On TopLink<sup>33</sup> (Shaun Smith)\*  
EclipseLink<sup>34</sup>\* Hibernate Blog<sup>35</sup>



#### 55.0.5 Products

\* Oracle TopLink Home<sup>25</sup>\*  
EclipseLink Home<sup>26</sup>\*  
TopLink Essentials  
Home<sup>27</sup>\* Hibernate  
Home<sup>28</sup>\* Open JPA  
Home<sup>29</sup>\* HiberObjects<sup>30</sup>

#### 55.0.7 Books

\* Pro EJB 3<sup>36</sup>

---

17 <http://forum.java.sun.com/forum.jspa?forumID=13>  
18 <http://saloon.javaranch.com/cgi-bin/ubb/ultimatebb.cgi?ubb=forum&f=78>  
19 <http://www.nabble.com/JPA-f27109.html>  
20 <http://www.nabble.com/EclipseLink-f26430.html>  
21 <http://www.eclipse.org/newsportal/thread.php?group=eclipse.rt.eclipselink>  
22 <http://forums.oracle.com/forums/forum.jspa?forumID=48>  
23 <http://forum.hibernate.org/>  
24 <http://www.nabble.com/java.net---glassfish-persistence-f13455.html>  
25 <http://www.oracle.com/technology/products/ias/toplink/index.html>  
26 <http://www.eclipse.org/eclipselink/>  
27 <https://glassfish.dev.java.net/javaee5/persistence/>  
28 <http://www.hibernate.org/>  
29 <http://openjpa.apache.org/>  
30 <http://objectgeneration.com/eclipse/>  
31 <http://java-persistence.blogspot.com/>  
32 <http://jroller.com/mkeith/>  
33 <http://ontoplink.blogspot.com/>  
34 <http://eclipselink.blogspot.com/>  
35 <http://blog.hibernate.org/>  
36 <http://www.amazon.com/gp/product/1590596455/102-2412923-9620152?v=glance&n=283155>

## 56 Contributors

Edits	User
3	Adrignola <sup>1</sup>
1	AllenZh <sup>2</sup>
1	Avicennasis <sup>3</sup>
1	Benjihawaii <sup>4</sup>
1	Bru <sup>5</sup>
1	Chrylis <sup>6</sup>
1	Cinhtau <sup>7</sup>
4	Cowwoc <sup>8</sup>
1	DanielSerodio <sup>9</sup>
2	David.shah <sup>10</sup>
1	Dirk Hünninger <sup>11</sup>
1	Djaquay <sup>12</sup>
1	Djclarke <sup>13</sup>
1	DragonxXx <sup>14</sup>
1	Dwersin <sup>15</sup>
1	Guruwons <sup>16</sup>
1	Inas66 <sup>17</sup>
1	Izogfif <sup>18</sup>
62	James.sutherland <sup>19</sup>
1272	Jamesssss <sup>20</sup>
1	Jan.derijke <sup>21</sup>

---

1	<a href="http://en.wikibooks.org/w/index.php?title=User:Adrignola">http://en.wikibooks.org/w/index.php?title=User:Adrignola</a>
2	<a href="http://en.wikibooks.org/w/index.php?title=User:AllenZh">http://en.wikibooks.org/w/index.php?title=User:AllenZh</a>
3	<a href="http://en.wikibooks.org/w/index.php?title=User:Avicennasis">http://en.wikibooks.org/w/index.php?title=User:Avicennasis</a>
4	<a href="http://en.wikibooks.org/w/index.php?title=User:Benjihawaii">http://en.wikibooks.org/w/index.php?title=User:Benjihawaii</a>
5	<a href="http://en.wikibooks.org/w/index.php?title=User:Bru">http://en.wikibooks.org/w/index.php?title=User:Bru</a>
6	<a href="http://en.wikibooks.org/w/index.php?title=User:Chrylis">http://en.wikibooks.org/w/index.php?title=User:Chrylis</a>
7	<a href="http://en.wikibooks.org/w/index.php?title=User:Cinhtau">http://en.wikibooks.org/w/index.php?title=User:Cinhtau</a>
8	<a href="http://en.wikibooks.org/w/index.php?title=User:Cowwoc">http://en.wikibooks.org/w/index.php?title=User:Cowwoc</a>
9	<a href="http://en.wikibooks.org/w/index.php?title=User:DanielSerodio">http://en.wikibooks.org/w/index.php?title=User:DanielSerodio</a>
10	<a href="http://en.wikibooks.org/w/index.php?title=User:David.shah">http://en.wikibooks.org/w/index.php?title=User:David.shah</a>
11	<a href="http://en.wikibooks.org/w/index.php?title=User:Dirk_H%C3%BCnniger">http://en.wikibooks.org/w/index.php?title=User:Dirk_H%C3%BCnniger</a>
12	<a href="http://en.wikibooks.org/w/index.php?title=User:Djaquay">http://en.wikibooks.org/w/index.php?title=User:Djaquay</a>
13	<a href="http://en.wikibooks.org/w/index.php?title=User:Djclarke">http://en.wikibooks.org/w/index.php?title=User:Djclarke</a>
14	<a href="http://en.wikibooks.org/w/index.php?title=User:DragonxXx">http://en.wikibooks.org/w/index.php?title=User:DragonxXx</a>
15	<a href="http://en.wikibooks.org/w/index.php?title=User:Dwersin">http://en.wikibooks.org/w/index.php?title=User:Dwersin</a>
16	<a href="http://en.wikibooks.org/w/index.php?title=User:Guruwons">http://en.wikibooks.org/w/index.php?title=User:Guruwons</a>
17	<a href="http://en.wikibooks.org/w/index.php?title=User:Inas66">http://en.wikibooks.org/w/index.php?title=User:Inas66</a>
18	<a href="http://en.wikibooks.org/w/index.php?title=User:Izogfif">http://en.wikibooks.org/w/index.php?title=User:Izogfif</a>
19	<a href="http://en.wikibooks.org/w/index.php?title=User:James.sutherland">http://en.wikibooks.org/w/index.php?title=User:James.sutherland</a>
20	<a href="http://en.wikibooks.org/w/index.php?title=User:Jamesssss">http://en.wikibooks.org/w/index.php?title=User:Jamesssss</a>
21	<a href="http://en.wikibooks.org/w/index.php?title=User:Jan.derijke">http://en.wikibooks.org/w/index.php?title=User:Jan.derijke</a>



1 Jomegat<sup>22</sup>  
1 Kencyber<sup>23</sup>  
1 Lawgers<sup>24</sup>  
8 Lmdavid<sup>25</sup>  
1 Moutonbreton<sup>26</sup>  
1 Panic2k4<sup>27</sup>  
1 Pengo<sup>28</sup>  
2 QuiteUnusual<sup>29</sup>  
1 Recent Runes<sup>30</sup>  
8 Sandro.roeder<sup>31</sup>  
1 Shaunmsmith<sup>32</sup>  
7 Svendhhh<sup>33</sup>  
3 Tewari.deepak<sup>34</sup>  
2 Theshowmecanuck<sup>35</sup>  
1 Tjoneill<sup>36</sup>  
1 Van der Hoorn<sup>37</sup>  
3 Vernetto<sup>38</sup>  
2 Vishal423<sup>39</sup>  
1 Webaware<sup>40</sup>  
1 Wutsje<sup>41</sup>  
1 YMS<sup>42</sup>  
1 ZeroOne<sup>43</sup>

---

22 <http://en.wikibooks.org/w/index.php?title=User:Jomegat>  
23 <http://en.wikibooks.org/w/index.php?title=User:Kencyber>  
24 <http://en.wikibooks.org/w/index.php?title=User:Lawgers>  
25 <http://en.wikibooks.org/w/index.php?title=User:Lmdavid>  
26 <http://en.wikibooks.org/w/index.php?title=User:Moutonbreton>  
27 <http://en.wikibooks.org/w/index.php?title=User:Panic2k4>  
28 <http://en.wikibooks.org/w/index.php?title=User:Pengo>  
29 <http://en.wikibooks.org/w/index.php?title=User:QuiteUnusual>  
30 [http://en.wikibooks.org/w/index.php?title=User:Recent\\_Runes](http://en.wikibooks.org/w/index.php?title=User:Recent_Runes)  
31 <http://en.wikibooks.org/w/index.php?title=User:Sandro.roeder>  
32 <http://en.wikibooks.org/w/index.php?title=User:Shaunmsmith>  
33 <http://en.wikibooks.org/w/index.php?title=User:Svendhhh>  
34 <http://en.wikibooks.org/w/index.php?title=User:Tewari.deepak>  
35 <http://en.wikibooks.org/w/index.php?title=User:Theshowmecanuck>  
36 <http://en.wikibooks.org/w/index.php?title=User:Tjoneill>  
37 [http://en.wikibooks.org/w/index.php?title=User:Van\\_der\\_Hoorn](http://en.wikibooks.org/w/index.php?title=User:Van_der_Hoorn)  
38 <http://en.wikibooks.org/w/index.php?title=User:Vernetto>  
39 <http://en.wikibooks.org/w/index.php?title=User:Vishal423>  
40 <http://en.wikibooks.org/w/index.php?title=User:Webaware>  
41 <http://en.wikibooks.org/w/index.php?title=User:Wutsje>  
42 <http://en.wikibooks.org/w/index.php?title=User:YMS>  
43 <http://en.wikibooks.org/w/index.php?title=User:ZeroOne>

# List of Figures

- GFDL: Gnu Free Documentation License. <http://www.gnu.org/licenses/fdl.html>
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. <http://creativecommons.org/licenses/by-sa/3.0/>
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. <http://creativecommons.org/licenses/by-sa/2.5/>
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. <http://creativecommons.org/licenses/by-sa/2.0/>
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. <http://creativecommons.org/licenses/by-sa/1.0/>
- cc-by-2.0: Creative Commons Attribution 2.0 License. <http://creativecommons.org/licenses/by/2.0/>
- cc-by-2.0: Creative Commons Attribution 2.0 License. <http://creativecommons.org/licenses/by/2.0/deed.en>
- cc-by-2.5: Creative Commons Attribution 2.5 License. <http://creativecommons.org/licenses/by/2.5/deed.en>
- cc-by-3.0: Creative Commons Attribution 3.0 License. <http://creativecommons.org/licenses/by/3.0/deed.en>
- GPL: GNU General Public License. <http://www.gnu.org/licenses/gpl-2.0.txt>
- LGPL: GNU Lesser General Public License. <http://www.gnu.org/licenses/lgpl.html>
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. <http://artlibre.org/licence/lal/de>
- CFR: Copyright free use.

- EPL: Eclipse Public License. <http://www.eclipse.org/org/documents/epl-v10.php>

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses<sup>44</sup>. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrowser.

---

<sup>44</sup> Chapter 57 on page 287

1		PD
2		
3		
4	Her Pegship <sup>45</sup>	LGPL
5		PD
6		PD
7	Jamesssss <sup>46</sup>	PD
8	Jamesssss <sup>47</sup>	PD
9	Jamesssss <sup>48</sup>	PD
10	Jamesssss <sup>49</sup>	PD
11	Jamesssss <sup>50</sup>	PD
12		PD
13		PD
14	Jamesssss <sup>51</sup>	PD
15	Jamesssss <sup>52</sup>	PD
16	Jamesssss <sup>53</sup>	PD
17		PD
18		
19		
20	Traced by User:Stannered <sup>54</sup> , original by David Vignoni	LGPL
21		

---

45 <http://en.wikibooks.org/wiki/User%3APegship>  
46 <http://en.wikibooks.org/wiki/User%3AJamesssss>  
47 <http://en.wikibooks.org/wiki/User%3AJamesssss>  
48 <http://en.wikibooks.org/wiki/User%3AJamesssss>  
49 <http://en.wikibooks.org/wiki/User%3AJamesssss>  
50 <http://en.wikibooks.org/wiki/User%3AJamesssss>  
51 <http://en.wikibooks.org/wiki/User%3AJamesssss>  
52 <http://en.wikibooks.org/wiki/User%3AJamesssss>  
53 <http://en.wikibooks.org/wiki/User%3AJamesssss>  
54 <http://en.wikibooks.org/wiki/User%3AStannered>



# 57 Licenses

## 57.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc.  
<<http://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer

network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sub-licensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

\* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. \* b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices". \* c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. \* d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

\* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. \* b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. \* c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. \* d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. \* e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial, or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

\* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or \* b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or \* c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or \* d) Limiting the use for publicity purposes of names of licensors or authors of the material; or \* e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or \* f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work)

from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10. 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

## 57.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <<http://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It implements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the exercise of, one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who may receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program. 13. Use with the GNU Affero General Public License.

authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero or more Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings created with vector drawing editors) and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal ef-

fect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year>  
<name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <<http://www.gnu.org/licenses/>>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author>  
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <<http://www.gnu.org/licenses/>>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <<http://www.gnu.org/philosophy/why-not-lgpl.html>>.

PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties; any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you

\* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. \* B. List on the Title

Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. \* C. State on the Title page the name of the publisher of the Modified Version, as the publisher. \* D. Preserve all the copyright notices of the Document. \* E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. \* F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. \* G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. \* H. Include an unaltered copy of this License. \* I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. \* J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. \* K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. \* L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. \* M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. \* N. Do not retile any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. \* O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add an-

other; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements". 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. TRANSLATION

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

\* a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or \* b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

\* a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. \* b) Accompany the object code with a copy of the GNU GPL and this license document.

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of

this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

## 57.3 GNU Lesser General Public License

### GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <<http://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below. 0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this license, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

### 4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

\* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. \* b) Accompany the Combined Work with a copy of the GNU GPL and this license document. \* c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. \* d) Do one of the following: o 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. o 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. \* e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

### 5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

\* a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. \* b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

### 6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.